Journal of Systems Architecture 140 (2023) 102894

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Online energy-efficient scheduling of DAG tasks on heterogeneous embedded platforms

Biao Hu [a], Xincheng Yang [a], Mingguo Zhao [b,*]

[a] College of Engineering, China Agricultural University, Beijing 100083, China
[b] Department of Automation, Tsinghua University, Beijing 100084, China

ARTICLE INFO

Keywords:
Task scheduling
DAG applications
Heterogeneous multi-processor systems
Energy minimization

ABSTRACT

Heterogeneous distributed computing systems are becoming more and more common to process practical applications in many embedded systems, where energy efficiency is very important, especially for battery-powered systems. This paper studies the problem of how to adaptively schedule dynamic applications on the fly on a heterogeneous embedded system such that the system's energy consumption can be minimized, while meeting applications' response time and reliability requirements simultaneously. We first study the problem of scheduling a set of static applications. This problem is formulated as an integer nonlinear programming problem by deriving the response time equation of directed acyclic graph (DAG) applications. A reliability-aware scheduling strategy is designed to solve this problem. On top of this strategy, we develop an algorithm that tries to use as few processors as possible to finish the computation because, in this way, the system's static energy can be saved. We then extend this solution to handle the problem of scheduling dynamic applications. By developing a task migration algorithm and a schedule adjustment algorithm, the system can adaptively handle suspended and newly released applications on the fly. Extensive simulation experiments are conducted, and the results demonstrate the high efficiency of our proposed approaches against some classic heuristics and a state-of-the-art approach.

1. Introduction

Background. The ever-evolving complexity of algorithms and massive data are placing higher and higher requirements on the performance of embedded systems. Traditional single-core embedded devices can no longer meet the computing power requirements of modern application scenarios. To meet the computation requirement, the multi-core processor architecture emerges as a popular computing paradigm that has been widely used in many embedded systems [1]. Compared to homogeneous processors, heterogeneous processors tend to perform better when dealing with an application with multiple computing requirements [2–4]. For instance, a floating-point calculation might be best handled by one core, while task switching might be best handled by another in a heterogeneous multi-core processor system. A typical example of heterogeneous processors is the i.MX 7 series of application processors. The i.MX 7 series utilize both the ARM Cortex-A7 and Cortex-M4 cores for general-purpose programmable processing. Its heterogeneous and asymmetric architecture provides the ultimate flexibility for customers by enabling a single-chip solution that can run sophisticated operating systems and provide real-time responsiveness.

With the continuous expansion in scale and function of an embedded system, its energy consumption is also increasing [5]. This poses big challenges for embedded systems, especially battery-powered embedded systems. In general, an embedded system works on a small-size computing board that has a limited size and power source. Designing a low-power embedded system can not only prolong its working hours but also reduce its operating cost and battery size. Additionally, a low-power system emits less energy and thus needs less cooling effort, which can further improve its working efficiency. However, the continuous increase in functional requirements and data volume makes embedded systems' architectures more and more heterogeneous, and the problem of performance degradation and energy shortage in the system becomes more severe than before. It is an important research topic to make full use of on-board computing resources in embedded systems to run programs with high energy efficiency.

Real-time computing capability and the system's reliability are two important performance aspects in embedded systems [6]. The task must not only be completed successfully but also be completed within the specified time; otherwise, the system will be regarded as a failure [7]. The specified time is the deadline for the task. In general, applications

* Corresponding author.
E-mail addresses: hubiao@cau.edu.cn (B. Hu), s20213071255@cau.edu.cn (X. Yang), mgzhao@mail.tsinghua.edu.cn (M. Zhao).

https://doi.org/10.1016/j.sysarc.2023.102894
Received 1 December 2022; Received in revised form 25 April 2023; Accepted 4 May 2023
Available online 10 May 2023

with real-time requirements mainly include: collecting data from the outside, requiring processing and responding to information at a certain moment or time; running in sequence according to programming instructions, where each task needs to be completed before the next task starts. The operating system can use some scheduling algorithms to meet the real-time requirements of tasks and allocate resources reasonably, such as first-come, first-served scheduling algorithms, least slack first algorithms, and earliest deadline scheduling algorithms. Real-time tasks can be classified into hard real-time tasks and soft real-time tasks. Their key difference is that a hard real-time system is one in which a single failure to meet the deadline may lead to a complete system failure, while a soft real-time system is one in which one or more failures to meet the deadline are not considered complete system failures but whose performance is considered degraded. In this paper, we only consider the timing requirements of hard real-time tasks for two reasons. First, hard real-time tasks have strict timing requirements and have a big impact on the system's safety, while soft real-time tasks do not. Thus, the scheduling strategy should be designed to guarantee the timing requirements of hard real-time tasks, and such requirements can be formulated as strict constraints. Second, because this paper is about reducing energy consumption, ignoring the performance of soft real-time tasks simplifies the problem and makes it easier to draw some conclusions.

Reliability refers to the probability of completing a specified function without error [8]. As the number of circuits inside the chip increases and the circuit size decreases, the chip's ability to resist the impact of high-energy particles is reduced. As a result, the probability of errors in the processor's execution of tasks increases, which reduces the system's reliability. Errors during system operation are generally divided into transient errors and permanent errors. The occurrence of transient errors is random, such as high-energy particles bombarding the chip and causing logic gate failure or memory bit flipping, etc. Such errors can disappear soon after they appear. Permanent errors may be caused by the failure of the logic device itself, such as disk damage, chip burnout, line breakage, etc. Permanent errors cannot be automatically eliminated after they occur. Transient errors are more common than permanent errors in a chip.

Motivation. From the perspective of the operating system, the system's performance in terms of energy efficiency, real-time computing capability, and reliability can be optimized by designing a good scheduling algorithm. Indeed, a plethora of scheduling approaches have been proposed to improve the system's performance on the three aspects. Scheduling given periodic tasks in a heterogeneous system has been proven to be an NP-hard problem in [9]. Scheduling applications is often more complicated because an application consists of many tasks that should be assigned to specified processors. List scheduling is a popular technique to schedule applications in a heterogeneous system. In general, an application can be modeled as a DAG, where a list scheduling algorithm first decides an assignment order for the application's tasks and then assigns them successively to processors in a greedy way. For example, in [10–12,12–15], the system's energy-efficiency, real-time computing capability, and reliability are jointly taken into account when designing a list scheduling algorithm. These approaches either study the problem of scheduling a single DAG application or make the assumption that DAG applications are not changed at runtime. None of them can be used to schedule dynamic DAG applications, with the key difference being that the former releases its DAG workload at a constant frequency and cannot be suspended or re-started at runtime, whereas the latter can be suspended and re-started at runtime. In reality, as some functions in an embedded system are turned on or off, active applications change, and the scheduler must make some adjustments based on the system's runtime state to process them better.

Contributions. This paper proposes an on-the-fly energy-efficient scheduling approach for dynamic DAG applications on a heterogeneous embedded system to reduce system energy consumption. Specifically, we have made the following contributions:

- This scheduling problem is mathematically formulated as an integer nonlinear programming problem with the aim of minimizing its energy consumption and with applications' response time and reliability as constraints.
- The schedule for static applications is built by decomposing DAG applications into tasks according to the upward rank value, and a reliability-aware scheduling strategy is proposed to successively assign each task to an appropriate processor. A processor-merging algorithm is then developed to shut down as many processors as possible to save energy.
- We create an online task migration algorithm for dynamically changing applications to safely migrate tasks without violating their real-time and reliability requirements.
- Extensive experiments are conducted to evaluate the performance of schedulability and energy consumption for handling both static and dynamic applications. The results show that our proposed solution has higher schedulability and lower energy consumption compared to some other classic heuristics and a state-of-the-art processor merging approach.

The remainder of this paper is structured as follows: Section 2 reviews the related work. Section 3 presents the models, including the system's architecture, reliability, power, and applications running on them. Section 4 presents the mathematical formulation of this scheduling problem. Section 5 presents the energy-efficient scheduling solution for a given set of static applications, and Section 6 presents the adaptive scheduling scheme for dynamic applications. Experimental results are presented in Section 7. The conclusion of this paper is provided in Section 8.

2. Related work

Because of the rapid development of IoT devices, embedded systems with high computing power and energy efficiency are required. Driven by this demand, embedded systems often adopt the heterogeneous multiprocessor computing architecture due to its powerful computing capability. How to design a schedule plays an important role in the effectiveness of running applications on this platform, and a plethora of scheduling techniques have been proposed to design a high-performance schedule. Here we review related works on scheduling tasks and static and dynamic applications on a heterogeneous multiprocessor system.

In general, the scheduling techniques in a multiprocessor system can be classified into global scheduling and partitioned scheduling, where the former decides the task assignment at runtime and the latter decides the task assignment in advance, and this assignment is not changed at runtime [16]. Task assignment with deadline constraints in a multiprocessor system has been demonstrated to be an NP-hard problem [17]. As energy consumption and reliability are two important aspects, some scheduling approaches are proposed to optimize the system's performance based on these two aspects. Two energy-aware data allocation and task scheduling heuristics are proposed in [18] to solve the problem of task scheduling combined with data allocation and energy consumption on heterogeneous distributed shared-memory multiprocessor systems. Analogously, the task scheduling and data allocation problems are discussed in [19] to minimize the total energy consumption, and two algorithms are proposed to solve this problem. The two works differ from this one in that they focus on how to share memory while ignoring the requirement for reliability. A reliability-aware task scheduling approach based on the whale optimization algorithm is proposed in [20] to improve the energy efficiency of a heterogeneous multiprocessor system. The whale optimization is a recently popular meta-heuristic algorithm that relies on heavy search computation. To schedule a mixed set of tasks, including both time-

and event-driven tasks, a real-time scheduling methodology based on satisfiability modulo theories is proposed in [21]. This work is limited to meeting real-time performance requirements and does not provide a solution for minimizing energy consumption.

Some tasks in many embedded systems are not separated but linked in a specific structure. Directed acyclic graph representations of partial orderings have many applications in scheduling for systems of tasks with ordering constraints. A BH-Mixed scheduling algorithm for DAG tasks with constrained deadlines is proposed in [22] that combines the partitioned algorithm, the federated scheduling algorithm and the GFP algorithm. In [23], an energy-aware scheduler is developed to schedule applications on edge devices to minimize the latency caused by re-timing without compromising on energy efficiency. A two-stage scheme is proposed in [24] to allocate and schedule dependent tasks with energy and real-time constraints on heterogeneous MPSoC systems to maximize system quality of security. One of its contributions is to model this problem as a mixed-integer linear problem. However, its solution cannot be used to solve our studied problem for three reasons. First, its workload model is a dependent task model, not as complex as the DAG application model. Second, all its tasks share the same deadline, instead of being constrained by different deadlines. Third, its objective is to maximize the system's security instead of minimizing energy consumption. Unlike [24], Huang et al. study how to achieve the best performance with reliability, energy, and makespan constraints in depth in [25–27]. These works, however, primarily focus on tuning the processor's running frequency for each task in a DAG application, which differs from our studied problem, which involves multiple dynamic applications whose tasks must be assigned to a set of heterogeneous processors. In [28], a method is proposed to assign cores for heavy tasks and a response time is proposed for the light tasks. In [29], a novel bi-objective genetic algorithm is presented to pursue low energy consumption and high system reliability. In [30], an improved hybrid version of the chemical reaction optimization (CRO) method is developed for solving the DAG-based task scheduling problem. Both of these works are aimed at scheduling a single static DAG application. An interesting work is presented in [31] that presents efficient distributed approaches to core maintenance on large dynamic graphs. In [10], two task scheduling algorithms called maximum reliability standby-sparing and dynamic reallocation fault-tolerant use dynamic power management and dynamic voltage frequency scaling techniques to improve energy efficiency and meet the reliability requirement. The optimal frequency choice for each task in an application with a reliability requirement is also studied in [6,32–35]. However, all these approaches only target either a single DAG application or some static applications.

Except for static applications whose arrival time has been known before designing a schedule, dynamic applications that arrive dynamically are also very common in embedded systems. To schedule dynamic applications, the designed schedule should be able to make task assignment decisions online to improve the system's performance. This problem is discussed in [25,36], where the main scheduling objectives are reliability, energy consumption, and makespan, and a search-based optimal energy allocation combined out-degree scheduling is proposed to optimize the three scheduling objectives. The list scheduling scheme is a common solution to schedule dynamic applications. Some effective heuristics have been proposed by Xie et al. to design a list schedule with the aim of reducing energy consumption, improving reliability, and minimizing makespan [37–40]. The problem with these list scheduling approaches is that they are not able to provide a sufficient guarantee on the application's response time and reliability. In contrast, our proposed approaches overcome this problem by deriving a sufficient bound on the application's response time and reliability.

3. Related models

In this section, we first introduce a system model, and then introduce related reliability and power models.

Table 1
Main notations in this study.

Notation	Definition
m	server count in the system
n	application count in the system
G_i	ith DAG application
P_h	hth processor
T_j^i	jth task of G_i
$e_{j,k}^i$	edge between T_j^i and T_k^i
$w_{j,h}^i$	computation time of task T_j^i running on processor P_h
f_i	release frequency of G_i
R_i	reliability requirement of G_i
D_i	deadline of G_i
$\|\mathcal{T}_i\|$	task count of $\mathcal{T}_i$
λ_h	failure rate of processor P_h
Y_S	system's energy consumption
$x_{j,h}^i$	assignment of T_j^i to processor P_h
o_j^i	priority of task T_j^i
st_j^i	release time of v_j^i
rt_j^i	response time of v_j^i
ft_j^i	finish time of v_j^i
$rank_u(T_j^i)$	upward rank value of T_j^i

3.1. System model

The distributed computing platform can be generalized as a model consisting of a set of heterogeneous processors that provide different computing capabilities [41–43]. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$ denote a set of m heterogeneous processors. At the beginning, there are some DAG applications that need to be scheduled. We denote them as $\mathcal{G} = \{G_1, G_2, \ldots, G_n\}$. Each DAG application G_i is associated with $(\mathcal{T}_i, E_i, W_i, f_i, R_i, D_i)$ where $\mathcal{T}_i$ is the set of its inside tasks; E_i is the set of those task dependencies; and W_i is the task execution time matrix; f_i is its release frequency; R_i is its reliability requirement per execution of G_i; D_i is its response time requirement. In particular, $\mathcal{T}_i = \{T_1^i, T_2^i, \ldots, T_N^i\}$ where a vertex T_j^i represents the jth task of this application. E_i is a $|\mathcal{T}_i| \times |\mathcal{T}_i|$ communication cost matrix, where $e_{j,k}^i$ denotes the dependency and communication cost between task T_j and T_k. For the simplicity, we use $|X|$ to stand for the set size of X. $e_{j,k}^i > 0$ if T_j^i must be completed ahead of T_k^i, and $e_{j,k}^i = 0$ otherwise. W_i is a $|\mathcal{T}_i| \times |\mathcal{P}|$ computation cost matrix, where $w_{j,h}^i \in W_i$ denotes the computation time of task T_j^i running on processor P_h.

The entry and exit tasks are two special tasks in a DAG application, where the former has no predecessor and the latter has no successor. If a DAG has multiple entry or exit tasks, a dummy entry or exit task with zero-weight dependencies can be added to the task graph so that there is one entry and one exit only. We denote the first task as an entry task and the last task as an exit task, i.e., $T_1^i = T_{\text{entry}}^i$ and $T_N^i = T_{\text{exit}}^i$. The meanings of important symbols have been summarized in Table 1.

Note that, with the time going on, the DAG application set in the system needs to be updated because some applications may be suspended and some new applications may be started. The designed schedule thus needs to be adaptively adjusted to meet the service requirements of newly released DAG applications.

3.2. Reliability model

The reliability of a system is defined as the probability that it can continuously work without failure during a given time interval [44]. In general, the failures of a system include hardware failures and software failures, where hardware failures refer to a hardware component stopping its designed function and software failures refer to the occurrence of an incorrect output caused by design errors. Software errors are unrelated to hardware components. In this work, it is assumed that software is very well designed and that failure does not occur. The occurrence of hardware failures is assumed to be transient, and it follows a Poisson probability distribution [45]. Such an assumption has

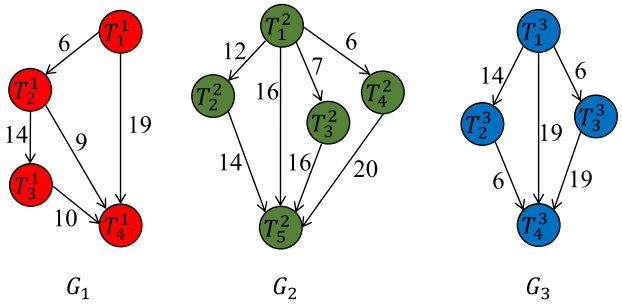


Fig. 1. Motivating example of three DAG-based applications scheduled in a system with three heterogeneous processors.

been widely used, e.g., [46–53]. Then, the reliability can be denoted as $e^{-\lambda\Delta}$, where Δ is the time interval and λ is the constant failure rate per time unit of a processor. For processor P_h of a heterogeneous system, we use λ_h to denote its failure rate. Then the execution time of a task T_j^i running on P_h is $w_{j,h}^i$, and its reliability is

$$R(T_j^i) = \mathrm{e}^{-\lambda_h \cdot w_{j,h}^i}. \tag{1}$$

Because an application does not fail only when all its tasks do not fail, the reliability of G_i for one release is thus

$$R(G_i) = \prod_{1 \le j \le |G_i|} R(T_j^i). \tag{2}$$

3.3. Power model

For the technique of dynamic power management, every processor has three modes, i.e., *sleep*, *standby*, and *active*. A processor is in *active* mode when it is executing a task and in *standby* mode when it is ready to execute a task. The *sleep* mode denotes that a processor is neither able to execute any task nor provide a buffer to store any task execution. However, the *standby* mode and *sleep* mode can be switched back and forth with some extra energy consumption. Here we adopt a power model that has been widely accepted in many other related works, such as [6,12,14,54]. In this model, the power of a processor P_h in *active* mode can be represented as

$$\Omega_h^a = \Omega_h^s + \Omega_h^d = \Omega_h^s + \beta_h \cdot f_h^m \tag{3}$$

where Ω_h^s is the processor's static power, which is also its power in *standby* mode; Ω_h^d is the processor's dynamic power, which is the product of a circuit-dependent constant β_h and the processor's frequency f_h raised to the m power. The power of processor's *sleep* mode is denoted as Ω_h^σ. Here, we have $\Omega_h^a > \Omega_h^s > \Omega_h^\sigma$. Then, we can get the energy consumption of a processor after working for a certain length of time. As an example, we assume that processor P_h works for a time length of Δ, in which it is in *active* mode of executing tasks for a time length of Δ_1, in *standby* mode ready for executing tasks for a time length of Δ_2, in *sleep* mode for a time length of Δ_3. The mode-switch between *standby* and *sleep* happens q times. Its energy consumption is thus

$$Y(P_h) = \Omega_h^a \cdot \Delta_1 + \Omega_h^s \cdot \Delta_2 + \Omega_h^\sigma \cdot \Delta_3 + q \cdot (E_{s\sigma} + E_{\sigma s}). \tag{4}$$

where $\Delta = \Delta_1 + \Delta_2 + \Delta_3$, and $E_{s\sigma}$ and $E_{\sigma s}$ denote the energy overhead for the mode-switch from *standby* to *sleep* and back.

The system's energy consumption is the sum of energy consumption from all processors, i.e.,

$$Y_S = \sum_{h=1}^{m} Y(P_h).$$

Example 1. We use an example to explain the system model [55]. The system consists of three heterogeneous processors, and there are three DAG applications, as shown in Fig. 1, where the edge number represents the communication cost. It is assumed that a task can be processed by any processor. Each application has a specific deadline and reliability requirement. Because processors have different computing capabilities, the computation time of a task will vary across processors, as presented in Table 2. Processors' failure rates are $9 \cdot 10^{-5}$, $6 \cdot 10^{-5}$, $6 \cdot 10^{-5}$, respectively. This example presents the problem of scheduling three DAG-based applications in a distributed system with three heterogeneous processors.

4. Problem

Our studied problem is how to design a schedule for a set of DAG applications and how to adjust this schedule when this DAG application set changes at runtime, such that the system's energy consumption can be minimized. In this section, we present how to formally formulate the problem of designing a schedule for a given set of DAG applications as an integer program with nonlinear constraints. The schedule adjustment for the changing $\mathcal{G}$ on the fly is based on this initial schedule.

4.1. Input, output, and objective

The input includes the hardware platform denoted as $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$, and a set of safety-critical DAG applications denoted as $\mathcal{G} = \{G_1, G_2, \ldots, G_n\}$.

To schedule tasks in a processor, we use the non-preemptive fixed-priority policy because it is common in embedded systems due to its stability and lightweight. Thus, except for the task assignment, a specific priority must be decided for any task. We use a tuple $\chi_j^i = \{\mathbf{x_j^i}, o_j^i\}$ to represent the two assignments, where $\mathbf{x_j^i} = [x_{j,1}^i, x_{j,2}^i, \ldots, x_{j,m}^i]$. Specifically, $x_{j,h}^i = 1$ if T_j^i is assigned to hth processor. Otherwise $x_{j,h}^i = 0$. o_j^i denotes this task's priority. For each task in a DAG application, the outputs should give the task assignment and the priority in its assigned processor. Thus, output variables including $x_{j,h}^i$ and o_j^i are integers.

The optimization objective is to minimize the system's energy consumption, i.e.,

$$\min Y_S.$$

4.2. Constraints

The constraints generally come from the task assignment, reliability requirement, and response time requirement.

4.2.1. Task assignment

Because a task can be assigned to a processor only, we have a linear constraint

$$\sum_{h=1}^{m} x_{j,h}^i = 1, \ \forall i = 1, \ldots, n, \ j = 1, \ldots, |\mathcal{T}_i|$$

where $x_{j,h}^i \in \{0, 1\}$.

4.2.2. Reliability requirement

We denote $\Lambda = [\lambda_1, \lambda_2, \ldots, \lambda_m]$ as the failure rate vector of processors from P_1 to P_m. Accordingly, we use $\mathbf{w_j^i} = [w_{j,1}^i, w_{j,2}^i, \ldots, w_{j,m}^i]$ to denote the execution time vector of T_j^i on processors from P_1 to P_m. Together with $\mathbf{x_j^i}$, we know that the reliability of T_j^i can be represented as nonlinear constraint

$$R(T_j^i) = \mathrm{e}^{-(\mathbf{x_j^i} \cdot \Lambda^{\mathrm{T}}) \cdot (\mathbf{x_j^i} \cdot \mathbf{w_j^i}^{\mathrm{T}})}, \tag{5}$$

where $\mathbf{x_j^i} \cdot \Lambda^T$ is the failure rate of assigned processor, and $\mathbf{x_j^i} \cdot \mathbf{w_j^i}^T$ is the execution time of T_j^i on the assigned processor. By (2), we know that the reliability requirement is

$$R(G_i) = \prod_{1 \le j \le |\mathcal{T}_i|} R(T_j^i) \ge R_i.$$

Table 2
Specifications of the three DAG-based applications that are processed in the three processors.

$G_1.deadline = 225,\ G_1.reliability = 0.991$				
Tasks	T_1^1	T_2^1	T_3^1	T_4^1
P_1	17	19	24	12
P_2	18	15	15	21
P_3	11	23	15	21
$rank_u$	101	79	46	18

$G_2.deadline = 215,\ G_2.reliability = 0.979$					
Tasks	T_1^2	T_2^2	T_3^2	T_4^2	T_5^2
P_1	20	21	17	24	22
P_2	24	16	20	16	13
P_3	13	21	16	21	21
$rank_u$	85	53	53	60	19

$G_3.deadline = 200,\ G_3.reliability = 0.983$				
Tasks	T_1^3	T_2^3	T_3^3	T_4^3
P_1	18	14	20	11
P_2	11	22	15	24
P_3	12	24	15	14
$rank_u$	73	43	53	17

	λ_h	Ω_h^a	Ω_h^s	Ω_h^σ	$E_{s\sigma}$	$E_{\sigma s}$
P_1	$9 \cdot 10^{-6}$	8	4	1	2.5	1.5
P_2	$6 \cdot 10^{-6}$	10	6	1	2.5	1.5
P_3	$6 \cdot 10^{-6}$	6	4	2	2.5	1.5

4.2.3. Response time requirement

Before we present the response time of an application, we have to analyze the response time of its inside tasks. We assume that processors stay in *active* mode continuously. Without loss of generality, we suppose that $x_{j,h}^i = 1$, i.e., task T_j^i is assigned to processor p_h. Its priority is o_j^i. We denote $\text{high}(T_j^i)$ and $\text{low}(T_j^i)$ stand for tasks whose priorities are higher and lower than T_j^i in this processor, respectively. From [56], we know that its response time rt_j^i in a system with non-preemptive execution scheme is

$$\begin{aligned} rt_j^i &= et_{\max} + rt_j^{i\prime} \\ et_{\max} &= \max_{T_k \in \text{low}(T_j^i)}\{et_k\} \\ rt_j^{i\prime} &= w_{j,h}^i + \sum_{T_k \in \text{high}(T_j^i)} \lceil rt_j^{i\prime} \cdot f^{T_k} \rceil \cdot et_k \end{aligned} \tag{6}$$

where $et_{\max}$ denotes the maximum interference from lower-priority tasks, and f^{T_k} denotes the release frequency of task T_k, and et_k denotes its execution time in this processor, and $\sum_{T_k \in \text{high}(T_j^i)} \lceil rt_j^{i\prime} \cdot f^{T_k} \rceil \cdot et_k$ is the execution interference from higher-priority tasks.

We then analyze the response time of a DAG application G_i, and use st_j^i, rt_j^i, ft_j^i to denote the task T_j^i's release time, response time and finish time, respectively. We assume that T_k^i has a higher priority than T_j^i ($o_k^i > o_j^i$) to get execution if they are in the same processor. According to sequence execution constraint, a task can be released only after all its predecessor and higher-priority tasks have been finished. Thus, it can be concluded that

$$\begin{aligned} st_j^i &= \max\left(ft_k^{ib}, st_z^{ib}\right), \\ ft_j^i &= st_j^i + rt_j^i \end{aligned} \tag{7}$$

where

$$\begin{aligned} ft_k^{ib} &= ft_k^i \cdot \mathbf{x_j^i} \cdot \mathbf{x_k^i}^{\mathrm{T}},\ \forall o_k^i > o_j^i \\ st_z^{ib} &= ft_z^i + e_{z,j}^i \cdot \left(1 - \mathbf{x_j^i} \cdot \mathbf{x_z^i}^{\mathrm{T}}\right),\ \forall e_{z,j}^i > 0. \end{aligned}$$

We know that $e_{j,z}^i > 0$ denotes that T_j^i is a predecessor task of T_z^i, and $\mathbf{x_j^i} \cdot \mathbf{x_z^i}^{\mathrm{T}} = 1$ only if both T_z^i and T_j^i are assigned to the same processor. By using (7) together with (6) for tasks from T_1^i to $T_{|\mathcal{T}_i|}^i$ successively, we can get the finish time of $T_{|\mathcal{T}_i|}^i$, which is the response time of this application.

4.3. Problem analysis

From the above analysis, we have formulated the problem of scheduling a set of safety-critical DAG applications on a heterogeneous distributed computing platform as an integer nonlinear programming (INLP) problem. To the best of our knowledge, there is not an effective optimization algorithm that can solve INLP quickly. This work aims to provide heuristic algorithms that can quickly find a schedule to meet all constraints. Besides, when the given DAG applications are updated at runtime, our proposed algorithm can quickly adjust this schedule to meet requirements of newly released DAG applications.

5. Scheduling a given $\mathcal{G}$

We first solve the scheduling problem when a set of DAG applications $\mathcal{G}$ are given at the beginning. Inspired by the heterogeneous earliest finish time (HEFT) algorithm [57], we decompose each DAG into a sequence of tasks by the upward rank value, and design a reliability-aware scheduling strategy to successively assign each task to an appropriate processor.

5.1. DAG decomposition

It has been proven in previous studies [57] that the upward rank value from HEFT algorithm is an effective approach to decide the task's assignment order in a DAG application for the aim of minimizing the response time. This value is defined as

$$rank_u(T_j^i) = w_j^i + \max_{e_{j,k}^i > 0}\{e_{j,k}^i + rank_u(w_k^i)\},$$

where $e_{j,k}^i > 0$ denotes that T_k^i is a successor task of T_j^i. Task execution sequence is ordered in the descending order of tasks' $rank_u$, i.e., the greater $rank_u$ is, the higher the execution priority of its corresponding task is. Without loss of generality, we suppose that T_j^i ($i = 1 \to n$, $j = 1 \to |\mathcal{T}_i|$) has already been ranked in this order.

5.2. Task assignments

Before assigning tasks, we have to ensure that the system has the capability to meet the reliability requirement of an application. We develop a reliability-aware scheduling scheme to meet such a requirement. First, by assigning each task of G_i to a processor that maximizes its reliability, we can get the maximum reliability of this system to run G_i, i.e.,

$$R_{\max}(G_i) = \prod_{1 \le j \le |\mathcal{T}_i|} R_{\max}(T_j^i) = \prod_{1 \le j \le |\mathcal{T}_i|} \max_{P_h \in \mathcal{P}} \mathrm{e}^{-\lambda_h \cdot w_{j,h}^i},\ \forall G_i \in \mathcal{G}.$$

If $R_{\max}(G_i) < R_i$, there does not exist a schedule that can meet the reliability requirement. After verifying that $R_{\max}(G_i) \ge R_i$, we have to carefully take the reliability requirement into account because any task misplacement could lead to the reliability violation. In order to maintain the reliability requirement and leave enough placement freedom for the current task that is ready to be assigned, all other unassigned tasks are assumed to be placed on the processor that maximizes its reliability. For example, a task T_j^i is ready to be assigned, where T_k^i ($1 \le k < j$) has been assigned, and T_z^i ($j < z < |\mathcal{T}_i|$) are assumed to be assigned to the processor that maximizes their reliability. Thus, we require that its minimum reliability should be greater than or equal to

$$R^{\text{req}}(T_j^i) = \frac{R_i}{\prod_{1 \le k < j} R(T_k^i) \cdot \prod_{j < z \le |\mathcal{T}_i|} R_{\max}(T_z^i)}. \tag{8}$$

Algorithm 1 Assignment Algorithm Towards T_j^i

Input: Applications $\mathcal{G}$, Processors $\mathcal{P}$, Task Assignments $\chi_b^a (1 \le a < i, 1 \le b \le |\mathcal{T}_a|)$, $\chi_k^i (1 \le k < j)$
Output: χ_j^i

$ft' = \inf$;
Get $R^{\text{req}}(T_j^i)$ by (8)
for $h = 1..m$ **do**
  for $o_j^i = 1..q_h + 1$ **do**
    $\mathbf{x_j^i} = \mathbf{0}$;
    $x_{j,h}^i = 1$;
    if $R(T_j^i)$ by (5) is greater than $R^{\text{req}}(T_j^i)$ **then**
      Update ft_j^i and $ft_{|\mathcal{T}_a|}^a$ ($\forall 1 \le a < i$) by (6), (7);
      if $ft_{|\mathcal{T}_a|}^a \le D_a$ & $ft_j^i < ft'$ **then**
        $ft' = ft_j^i$, $h' = h$, $o' = o_j^i$;
      end if
    end if
  end for
end for
if $ft' < \inf$ **then**
  $\mathbf{x_j^i} = \mathbf{0}$;
  $x_{j,h'}^i = 1$, $o_j^i = o'$;
else
  Failure;
end if

Algorithm 2 Initial Schedule Design

Input: Applications $\mathcal{G}$, Processors $\mathcal{P}$
Output: $\chi_b^a (1 \le a \le n, 1 \le b \le |\mathcal{T}_a|)$

$\mathcal{P}_{\text{active}}^{\text{temp}} = \mathcal{P}_{\text{active}} = \mathcal{P}$;
if Tasks are successfully assigned in $\mathcal{P}_{\text{active}}^{\text{temp}}$ **then**
  while true **do**
    Find out processors from $\mathcal{P}_{\text{active}}$ that meet (14), and store them to $\mathcal{P}_{\text{sleep}}^{\text{ready}}$, and sort $\mathcal{P}_{\text{sleep}}^{\text{ready}}$ in the descending order of $(\Omega_{h'}^a - \Omega_{h'}^s)$, $\forall P_{h'} \in \mathcal{P}_{\text{sleep}}^{\text{ready}}$;
    for $h' = 1..|\mathcal{P}_{\text{sleep}}^{\text{ready}}|$ **do**
      $\mathcal{P}_{\text{active}}^{\text{temp}} = \mathcal{P}_{\text{active}} \backslash \mathcal{P}_{\text{sleep}}^{\text{ready}}(h')$
      Decide new $\chi_b^a (1 \le a \le n, 1 \le b \le |\mathcal{T}_a|)$ by Algorithm 1;
      if All tasks are successfully assigned **then**
        $\mathcal{P}_{\text{active}} = \mathcal{P}_{\text{active}}^{\text{temp}}$
        break;
      else if $h' == |\mathcal{P}_{\text{sleep}}^{\text{ready}}|$ **then**
        return $\chi_b^a (1 \le a \le n, 1 \le b \le |\mathcal{T}_a|)$
      end if
    end for
  end while
else
  return failure;
end if

In this way, we transfer the reliability requirement from a DAG application to its each task.

In addition to the reliability requirement, our second aim is to meet the response time requirement of G_i. To achieve it, we let G_i's tasks finish as early as possible. We suppose that G_i's tasks are assigned in the order from $i = 1$ to $i = n$ successively. Now it turns to assigning G_i's task T_j^i and we presume that previous processor and priority assignments towards all G_k's tasks have met the reliability and response time requirement, i.e., $\chi_b^a (1 \le a < i, 1 \le b \le |\mathcal{T}_a|)$ are known. Besides, G_i's tasks $T_k^i (1 \le k < j)$ have also been assigned its processors and its priorities, $\chi_k^i (1 \le k < j)$ are also known. In this case, for the assignment of task T_j^i, there are m processor options, and in each processor there are $q_h + 1 (1 \le h \le m)$ priority options where q_h is the count of tasks that have been assigned on processor p_h. Together with the response time analysis of (6) and (7), we conclude that the best processor and priority option for T_j^i should be

$$\min \quad ft_j^i \tag{9}$$

$$\text{subject to} \quad 1 \le h \le m, \ 1 \le o_j^i \le q_h + 1 \tag{10}$$

$$R(T_j^i) \ge R^{\text{req}}(T_j^i) \tag{11}$$

$$(6), \ (7) \tag{12}$$

$$ft_{|\mathcal{T}_a|}^a \le D_a, \ \forall 1 \le a < i \tag{13}$$

where ft_j^i is the finish time of task T_j^i under constraints from (10) to (13). Our goal is to minimize ft_j^i. The constraint (10) provides the processor and priority option; (11) is the reliability requirement on T_j^i; (12) is the response time towards T_j^i; last but not the least, by (13), we guarantee that the assignment of T_j^i will not violate the response time requirement of previous assigned DAG applications. It can be found that constraint (10) is linear constraint, and other constraints are nonlinear.

We develop an algorithm as shown in Algorithm 1 to decide the assignment of processor and priority towards T_j^i. In general, we test every processor and priority assignment option towards T_j^i, and find the best option that meets all constraints and has the minimum time to finish the execution of T_j^i. This algorithm may also fail to find a feasible solution. Because there are $\sum_{i=1}^n |\mathcal{T}_i|$ tasks, the worst case is checking all priority options, i.e., $\sum_{i=1}^n |\mathcal{T}_i|$. Thus, its worst-case complexity is $O(\sum_{i=1}^n |\mathcal{T}_i|)$.

Example 2. Suppose that tasks from G_1, G_2, G_3 in Example 1 are successively assigned by Algorithm 1. When T_4^3 is ready to be assigned, other tasks have been assigned, as presented in Fig. 2. Since the achieved reliability of T_4^3 is greater than its required reliability in any processor, T_4^3 can be assigned to any one processor. After checking all its possible processor assignment and priority options, we find that its finish time is the least by assigning it to processor P_1. Thus, it will be assigned to processor P_1.

5.3. Energy-efficient scheduling

We ignore the energy consumption in order to simplify the problem of finding a schedule for a given $\mathcal{G}$ on a platform with m heterogeneous processors. However, when the system workload is light and the hardware resource is abundant, switching all processors to *active* is not energy-efficient. In this section, we discuss how to use as few processors as possible to serve the given $\mathcal{G}$ for the aim of saving energy.

We suppose that Algorithm 1 has successfully built a feasible schedule for the given $\mathcal{G}$ with m heterogeneous processors. Now we study how to switch some processors to *sleep* mode while satisfying the requirement of reliability and response time. If a processor $P_{h'}$ is switched to *sleep* mode, we should move all tasks on $P_{h'}$ to other processors. Besides, in order to guarantee the reliability requirement, we should make the maximum reliability greater than or equal to the required one for every G_i. That is

$$R_{\max}(G_i) = \prod_{1 \le j \le |\mathcal{T}_i|} \max_{P_h \in \{\mathcal{P} \backslash P_{h'}\}} e^{-\lambda_h \cdot w_{j,h}^i} \ge R_i, \ \forall G_i \in \mathcal{G}. \tag{14}$$

In this way, we can pick out the processors that are not necessary to meet the reliability requirement. Because the system should also meet response time requirement and minimize the energy consumption, we develop an algorithm as shown in Algorithm 2 to cover the two aspects. We first find out processors that are not necessary for the reliability requirement, and store them to $\mathcal{P}_{\text{sleep}}^{\text{ready}}$. From the energy consumption

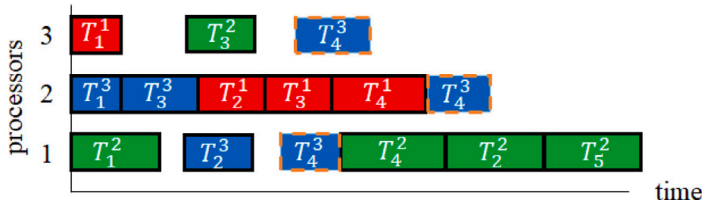


Fig. 2. The assignment options for T_4^3.

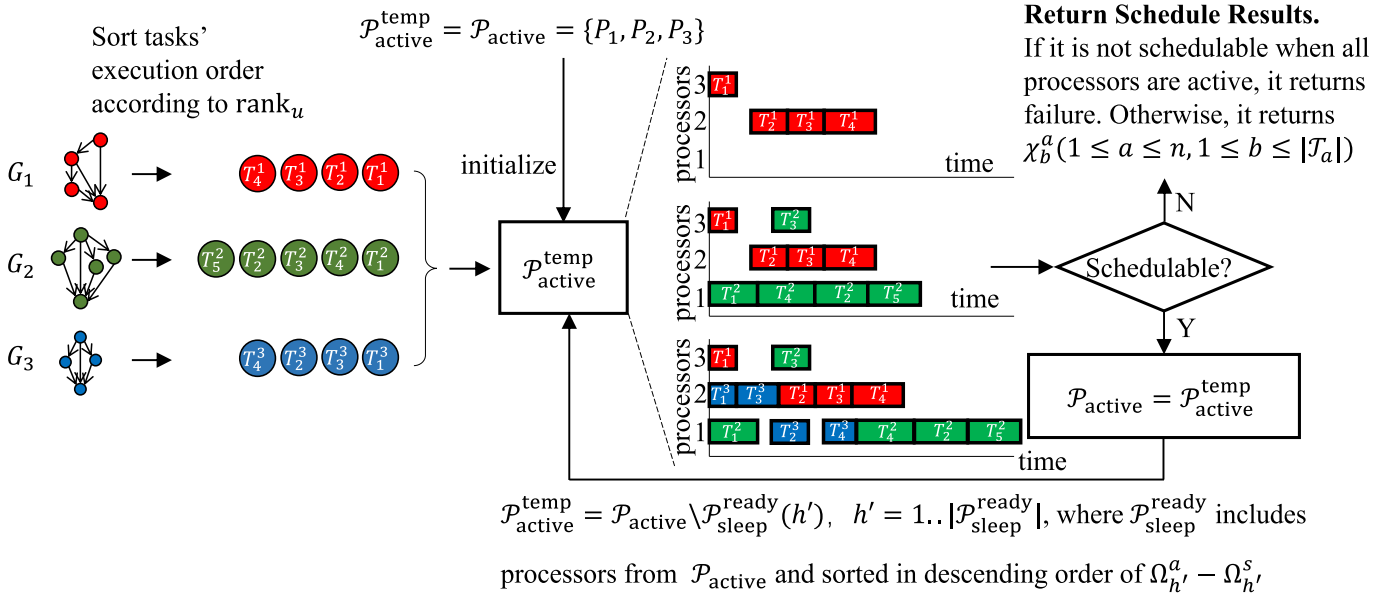


Fig. 3. The energy-efficient scheduling flowchart of the example.

of (4), we find that, if a processor $\mathcal{P}_{\text{sleep}}^{\text{ready}}(h')$ is switched to *sleep* for a time length of Δ, the energy reduction would be

$$\Delta E_{h'} = (\Omega_{h'}^a - \Omega_{h'}^\sigma) \cdot \Delta.$$

Thus, the greater $(\Omega_{h'}^a - \Omega_{h'}^\sigma)$ is, the more energy reduction would be. For this reason, we sort processors picked out by (14) in the descending order of $(\Omega_{h'}^a - \Omega_{h'}^\sigma)$. Then we test whether all tasks can be successfully assigned after $P_{h'}$ from $\mathcal{P}_{\text{sleep}}^{\text{ready}}$ is switched to *sleep*. If so, this processor is then switched to *sleep* mode. Otherwise, next processor is tested. This procedure continues until we cannot successfully find a processor without violating the response time requirement. Because processors are successively activated and Algorithm 1 is applied for assigning every task, the worst-case complexity is $O\left(m^2 \cdot (\sum_{i=1}^n |\mathcal{T}_i|)^2\right)$.

Example 3. The procedures of applying Algorithm 1 and Algorithm 2 to schedule applications in Example 1 are presented in Fig. 3. At the beginning, tasks' execution order is sorted according to their upward rank values, and tasks from G_1, G_2, G_3 are successively assigned to processors. Processors are assumed to be active to test whether G_1, G_2, G_3 are schedulable. If yes, processors are reordered according to their power of active mode and sleep mode. For this example, processors are reordered to $\{P_2, P_1, P_3\}$, and another set called $\mathcal{P}_{\text{sleep}}^{\text{ready}}$ is used to keep this order. Then, in the first round, P_2, P_1, P_3 are successively removed from $\mathcal{P}_{\text{active}}^{\text{temp}}$ to test whether the remained processors can schedule G_1, G_2, G_e3 or not. If not, it means that the three processors must be active to accommodate the three applications. If yes, one processor will be switched to sleep mode, and another round goes on to test whether it is still possible to switch another processor to sleep mode. The while loop breaks out when there is not one processor that can be switched to sleep mode. For this example, the three processors must be active.

6. Scheduling dynamic $\mathcal{G}$

We have presented how to design a schedule with as few processors as possible for a given $\mathcal{G}$. However, in many real-life embedded systems, applications are not continuously running. Some applications may be suspended and some new applications may be released at runtime. Hence, our scheduling algorithm should be adaptive to do some adjustment for the dynamic changing $\mathcal{G}$. In this section, we discuss how to adaptively schedule dynamic $\mathcal{G}$ for the aim of saving energy.

6.1. Applications suspended from $\mathcal{G}$

When some applications are suspended, the workload becomes less than before, leading to that the response time of remaining applications may also become less than before. For this reason, the workload reduction brings the possibility to switch some processors to *sleep* for the aim of saving energy. Here we discuss how to make some adjustment towards the previous schedule when the workload declines.

Unlike designing a completely new schedule for a given $\mathcal{G}$, switching a processor off at runtime only involves how to migrate tasks on this switched-off processor to other *active* processors. During this migration, it is important to not violate application's response time and reliability

Algorithm 3 Task Migration Algorithm Online

Input: Applications $\mathcal{G}$, Processors $\mathcal{P}$, Task Assignments χ_b^a ($1 \le a \le n$, $1 \le b \le |\mathcal{T}_a|$), Suspended Application G_κ
Output: χ_b^a ($1 \le a \le n$, $a \ne \kappa$, $1 \le b \le |\mathcal{T}_a|$)
1: $\chi_b^a = \chi_b^a \backslash \chi_b^\kappa$;
2: Find all processors that meet (15) and store them to $\mathcal{P}_{\text{sleep}}^{\text{temp}}$;
3: Sort $\mathcal{P}_{\text{sleep}}^{\text{temp}}$ in the descending order of $(\Omega_{h'}^a - \Omega_{h'}^\sigma)$, $\forall P_{h'} \in \mathcal{P}_{\text{sleep}}^{\text{temp}}$;
4: **for** $h' = 1..|\mathcal{P}_{\text{sleep}}^{\text{temp}}|$ **do**
5: **if** $P_{h'}$ is tested to be switched to *sleep* by Algorithm 4 **then**;
6: When an idle tick appears in every *active* processor, migrate tasks in $P_{h'}$ sequentially to other processors according to the result of Algorithm 4;
7: **end if**
8: **end for**

requirement. This is actually a complicated problem because any task migration has a big impact on all application's response time. Besides, the task migration should not interrupt the non-preemptive fixed-priority scheduling behavior in each processor. Here we propose a sequence migration algorithm that migrates tasks of an application in a strict timing sequence order.

First of all, picking up a processor to be switched off is the first step. According to the reliability requirement, the processor chosen to be switched off should still guarantee the application's reliability requirement. For example, if processor $P_{h'}$ is tested to be switched off, tasks on this processor are thus assumed to be migrated to processors that maximize their reliability. $P_{h'}$ can be possibly switched to *sleep* only if all applications' reliability under this assumption is greater than or equal to the required ones. Analogous to (14), we conclude this judgment to

$$R_{\max}(G_i) = \prod_{1 \le j \le |\mathcal{T}_i|} R(T_j^i) \ge R_i, \ \forall G_i \in \mathcal{G}. \tag{15}$$

where

$$R(T_j^i) = \begin{cases} e^{-\lambda_h \cdot w_{j,h}^i}, & \text{if } T_j^i \text{ is in } P_h \quad \text{and } P_h \ne P_{h'} \\ \max\limits_{P_h \in \{\mathcal{P} \backslash P_{h'}\}} e^{-\lambda_h \cdot w_{j,h}^i}, & \text{if } T_j^i \text{ is in } P_{h'} \end{cases}$$

We can then use (15) to find out processor candidate for switching to *sleep*. Candidates are sorted in the descending order of $(\Omega_{h'}^a - \Omega_{h'}^\sigma)$, and they are successively tested whether their tasks can be safely migrated to other processors without violating their response time requirement.

In order to derive an upper-bound of response time after a task migration, we restrict that the timing gap between two task migration should be large enough to let every processor have an idle tick within this gap. Because the idle tick indicates that tasks' execution before this tick has no effect on tasks' execution after it, the interference between two task migration is thus separated [58–60]. We can then analyze the effect of task migration on application's response time individually. We suppose that T_j^i is migrated from processor $P_{h'}$ to processor P_h. As presented in Section 4.2.3, we know that a task's response time is influenced by tasks only from its own processor. Thus, we only need to update the response time of tasks in processor $P_{h'}$ and P_h because there is not a change on tasks in other processors. After this update, we can get all remained applications' response time by (7). For a specific task, its migration and priority assignment are successively tested until success, i.e., applications' new response time and reliability met their requirement. If tasks on a processor can be successfully migrated to other processors, this processor can be switched to *sleep*. Such procedures are presented as the pseudo-code of Algorithm 3 and Algorithm 4. Analogous to the energy-efficient scheduling, we first find out all processor candidates that can be potentially switched to *sleep* without violating reliability requirement. Then, we try to switch them

Algorithm 4 Schedule Adjustment by Switching $P_{h'}$ to *Sleep*

Input: Applications $\mathcal{G}$, Processors $\mathcal{P}$, Task Assignments χ_b^a ($1 \le a \le n$, $1 \le b \le |\mathcal{T}_a|$), Suspended Application G_κ
Output: χ_b^a ($1 \le a \le n$, $a \ne \kappa$, $1 \le b \le |\mathcal{T}_a|$)
1: $\mathcal{P}_{\text{active}}^{\text{temp}} = \mathcal{P}_{\text{active}} \backslash P_{h'}$
2: $\mathcal{T}_{h'} \leftarrow$ tasks in $P_{h'}$
3: **for** $T_j^i \in \mathcal{T}_{h'}$ **do**
4: **for** $h = 1..|\mathcal{P}_{\text{active}}^{\text{temp}}|$ **do**
5: **for** $o_j^i = 1..q_h + 1$ **do**
6: $\mathbf{x}_j^i = \mathbf{0}$, $x_{j,h}^i = 1$;
7: Update all application's response time and reliability;
8: Break if requirement is met;
9: **end for**
10: **end for**
11: **end for**
12: **if** $\mathcal{P}_{h'}$ has no task **then**
13: $\mathcal{P}_{\text{sleep}} = \mathcal{P}_{\text{sleep}} \cup P_{h'}$;
14: Return χ_b^a ($1 \le a \le n$, $a \ne \kappa$, $1 \le b \le |\mathcal{T}_a|$)
15: **else**
16: Failure, $\mathcal{P}_{h'}$ cannot be switched to *sleep*;
17: **end if**

to *sleep* by testing whether tasks in them can be migrated to other *active* processors. Note that at runtime, tasks should be migrated one by one, and their migration timing gap should be large enough to let idle tick exist in every processor.

In Algorithm 4, there are $|\mathcal{T}_{h'}|$ tasks that need to be migrated out. For each migrated task, the number of place options that need to be checked is not greater than $\sum_{i=1}^n |\mathcal{T}_i|$. Thus, the worst-case complexity of Algorithm 4 is $O(|\mathcal{T}_{h'}| \cdot \sum_{i=1}^n |\mathcal{T}_i|)$. In Algorithm 3, there are m processors that could be turned to *sleep* mode, the worst-case complexity of Algorithm 3 is $O(m \cdot |\mathcal{T}_{h'}| \cdot \sum_{i=1}^n |\mathcal{T}_i|)$.

6.2. New applications start at runtime

We now discuss how to handle the case that some new applications may be released at runtime. Inspired by the scheduling algorithm for a given $\mathcal{G}$, the new schedule towards these newly released applications should be based on the previous schedule. We first try to assign new applications' tasks to the currently *active* processors. If such assignments are not successful, we switch another processor to *active*, and repeat assigning. Such procedures continue until success. Algorithm 2 can be used to search for a new schedule by switching more processors to *active*, instead of switching them to *sleep*.

We have presented 4 algorithms to schedule dynamic DAG applications with the aim of minimizing the energy consumption on the fly. Their relationship can be presented by a flowchart as shown in Fig. 4. In general, algorithms 1,2 are responsible for assigning tasks of new applications and algorithms 3,4 are responsible for migrating tasks when some applications are suspended.

Example 4. We suppose that G_2 is suspended after G_1, G_2, G_3 have been executed for a while following the designed schedule of Example 3. Then, tasks of G_2 will disappear in the schedule, as presented in Fig. 5. According to reliability requirement, P_1, P_2, P_3 are options that can be possibly switched into *sleep*. For energy consumption reasons, the order of such switches is P_2, P_1, P_3. Then, after switching P_2 into *sleep*, its tasks need to be migrated into P_1 or P_3. The final assignment is presented as Fig. 3, where T_3^1 is migrated into P_3, and T_1^3, T_3^3, T_2^1, T_4^1 are migrated into P_1. We find that it is not possible to switch P_1 or P_3 into *sleep* for the reason of meeting response time requirement.

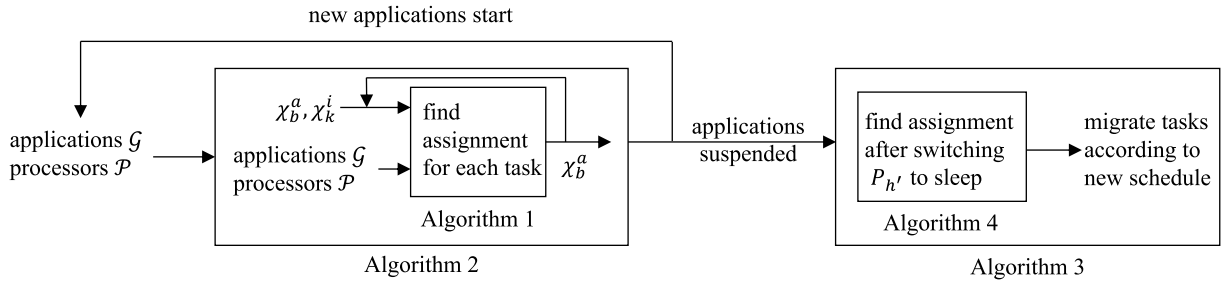


Fig. 4. The flowchart of using the presented 4 algorithms.

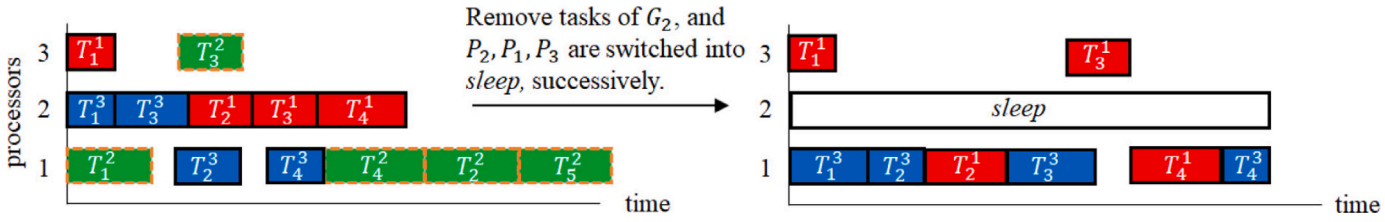


Fig. 5. The schedule results after G_2 is suspended.

7. Experimental evaluation

This section presents the performance comparison of our proposed approaches against some classic and state-of-the-art approaches. Based on the power specifications of i.MX 7 Dual Processors [61], the power consumption of a processor in *sleep* mode, *standby* mode and *active* mode is within [1, 2] W, [2, 6] W and [6, 10] W, respectively. The energy overhead of turning a processor from *standby* to *sleep* and back is $E_{\sigma s} = 1.5\,J$ and $E_{s\sigma} = 2.5\,J$. A processor's failure rate is within $[5, 10]\cdot 10^{-5}$. It should be noted that every processor has its own distinct power and reliability characteristics. The DAG structure is generated by the tgff tool [62]. The computation time of a task on different processors is different, but it is within [0.1, 0.4] seconds. The communication overhead between two tasks on different processors is within [0.01, 0.1] seconds. Other recent works, such as [6,12,33], provide data for DAG applications.

7.1. Compared approaches

We have presented an energy-efficient scheduling approach to schedule DAG tasks. Since there is not an existing approach that solves our studied problem, we take some ideas from previous works that have similar settings and modify their algorithms to solve our problem. In general, the compared approaches for performance evaluation are listed below.

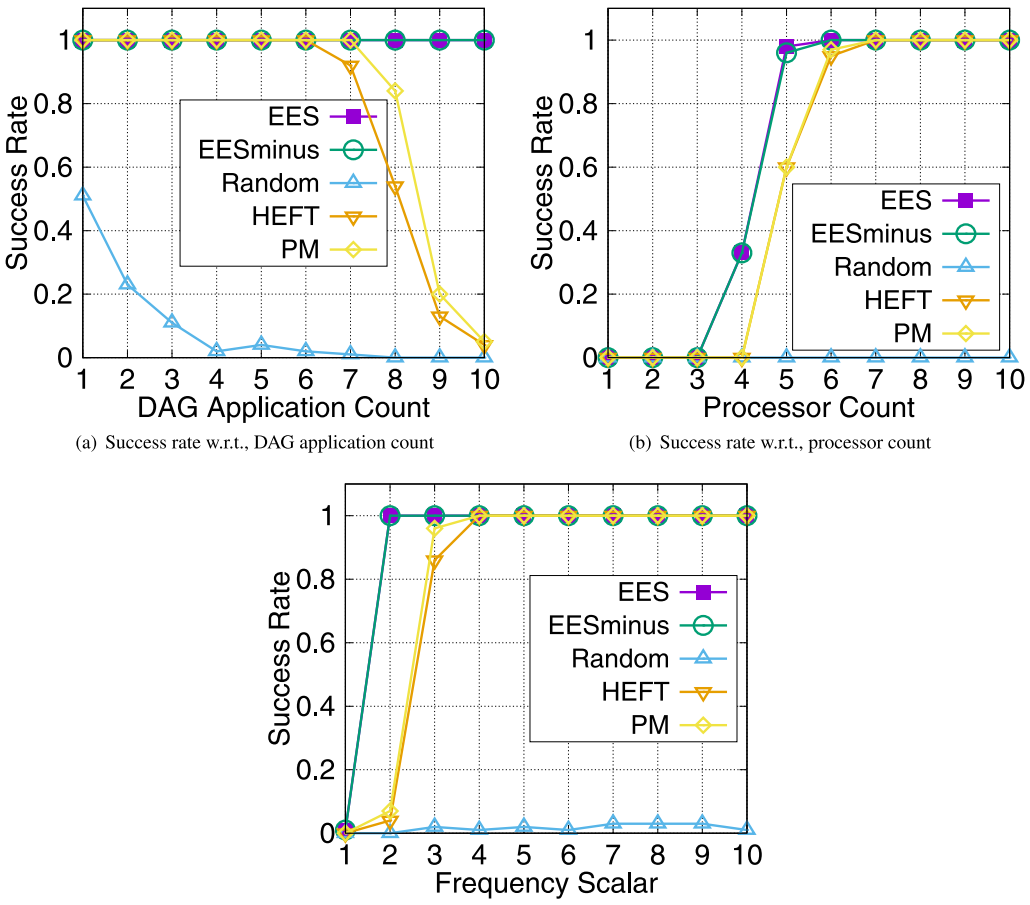
1. EES: This refers to our proposed energy-efficient scheduling approach that tries to use as few processors as possible to run all DAG applications, such that the system's energy consumption can be minimized.
2. EESminus: This approach decides the task's assignment and priority in the same way as EES. However, unlike EES, this approach does not turn any processor to *sleep* mode, which means it does not aim to reduce the power consumption on purpose.
3. Random: This refers to assign DAG's tasks to a processor and decides its execution priority randomly.
4. HEFT: This approach first decides the task assignment and priority by the algorithm of heterogeneous earliest first time towards each individual DAG [57]. Then, in the ascending order of application's deadline, tasks are successively assigned to the corresponding processors, where later tasks have lower priorities than formers.
5. PM: This approach refers to the processor-merging algorithm proposed in [6]. It takes the requirement of reliability and response time into account when designing a schedule. However, it only works on a single DAG application. Here we modify it to group all DAG applications into one, and apply PM algorithm to design a schedule for them.

For the comparison, two metrics are used: the success rate and energy consumption. The former is the ratio of successful schedulability tests among all tests, and it represents the schedulability of an approach to successfully assign DAG's tasks, and the latter represents the energy-efficiency of the designed schedules.

7.2. Simulation results of static DAG applications

We investigate the effect of DAG application count, processor count, and application release frequency on the schedulability test performance of the compared approaches. Here we assume that the system has only static DAG applications that will not be suspended at runtime, and new applications will not be released again. For every parameter configuration, we perform 100 experiments on randomly generated DAG applications and use the average success rate to represent their schedulability and average energy consumption to represent their power efficiency. The length of the simulation is 1 h.

Effect of DAG Application Count. In this simulation, we vary the DAG application count from 1 to 10 by setting the processor count to 10. An application's release frequency is the reciprocal of all its tasks' execution time divided by a scalar of 5. The result of success rate is presented in Fig. 6(a), where we find that EES and EESminus have the highest success rate among all approaches, which is 100% success among all tests. The success rate of PM is lower than EES and ESSminus, but higher than HEFT and Random. It can be found that the success rate of PM drops to 0 when the application count reaches 10 or more. HEFT loses to PM slightly because PM is an approach that takes advantage of HEFT to shorten its response time. Random performs the worst because it does not take any requirements or optimization objectives into account. Fig. 7(a) depicts the energy consumption of all compared approaches with varying DAG application counts, with EES being the best and PM being the second-best. Other approaches perform very closely in terms of energy consumption. The reason for the good performance of EES and PM is that the two approaches switch unnecessary processors to *sleep* mode to reduce energy consumption. Such an energy reduction becomes less when the application count



(a) Success rate w.r.t., DAG application count

(b) Success rate w.r.t., processor count

(c) Success rate w.r.t., frequency scalar

Fig. 6. Success rate results of compared approaches with regard to different parameters.

increases, because more applications need more processors to stay *active* to execute them.

Effect of Processor Count. In this simulation, we vary the processor count from 1 to 10 by setting the application count to 5. An application's release frequency is the reciprocal of all its tasks' execution time divided by a scalar of 5. The success rates of all compared approaches with varying processor counts are presented in Fig. 6(b), where we find that the approaches with the highest success rates are still EES and EESminus. HEFT and PM perform very closely. The success rate of random assignment is close to zero, indicating that random assignment is difficult to meet the system's response time and reliability requirements. The results of their energy consumption are presented in Fig. 7(b), where the energy consumption corresponding to a processor count of $1, 2, 3$ is missing, because all approaches fail to find a feasible schedule when the processor count is less than 3. EES has the least energy consumption, and PM has a slightly higher energy consumption than EES. Other approaches have much higher energy consumption than EES and PM, and the gap between their energy consumption increases with increasing processor count. In addition, we find that with the increased processor count, the success rate of all approaches and the energy efficiency of EES and PM increase. This is because more processors mean more computing power, which increases the system's schedulability and EES's and PM's chances to switch processors to *sleep* mode to save energy.

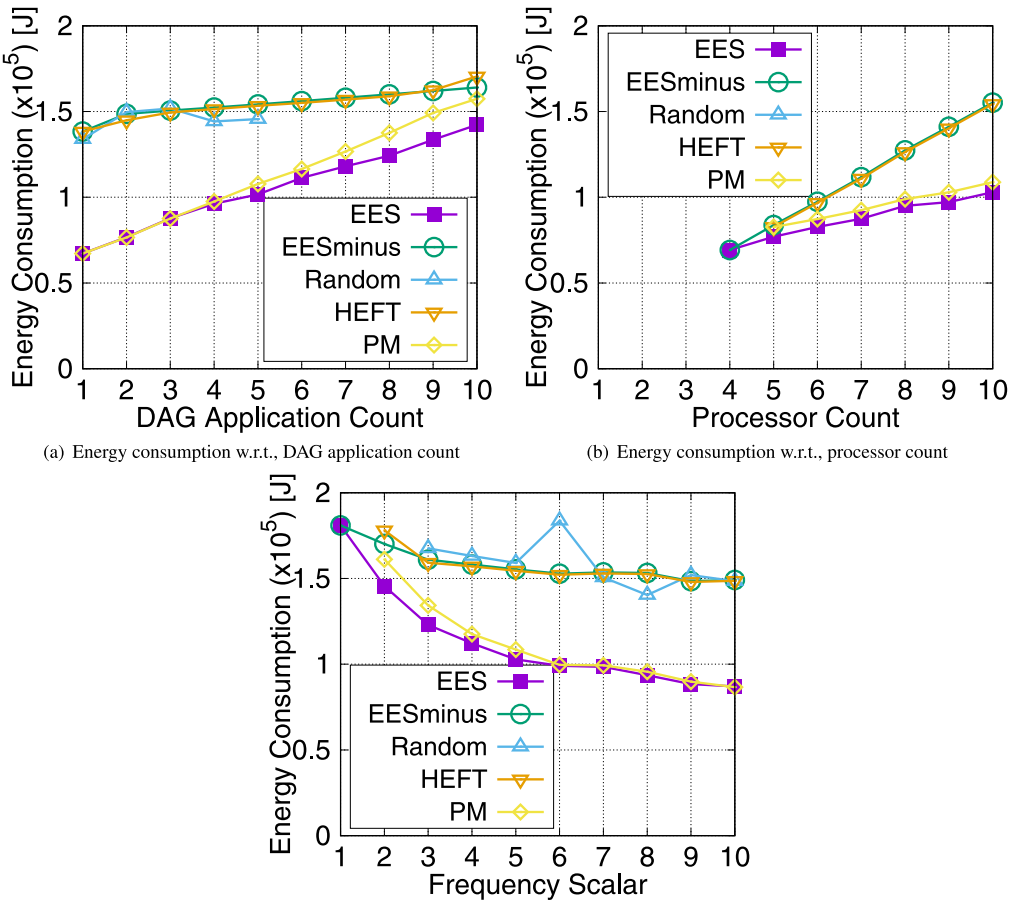
Effect of Application's Release Frequency. In this simulation, the scalar used for defining the application's release frequency above varies from 1 to 10. The processor count is set to 10 and the application count is set to 5. The success rate of all compared approaches is shown in Fig. 6(c), with EES and EESminus having a higher success rate than others. With the increase of the frequency scalar, the success rate of all approaches increases. The reason for this phenomenon is that application frequency decreases with the increase of the frequency scalar, and there would be less workload in the system. Hence, the system has a higher success rate of scheduling them. The energy consumption of all compared approaches in terms of increasing frequency scalar is shown in Fig. 7(c), and the two best approaches remain EES and PM. Their superiority in reducing energy consumption becomes greater with the increase of the frequency scalar.

In general, from the above experimental results, we find that the approach with the highest success rate and the least energy consumption in different parameter configurations is our proposed EES approach. EESminus has the same success rate as EES, but its energy consumption is as bad as HEFT and Random. PM has energy consumption very close to EES, but its success rate is as bad as HEFT.

7.3. Simulation results of dynamic DAG applications

In this simulation setting, DAG applications are assumed to be dynamically released or suspended at runtime. There are 10 DAG applications running on a system with 10 processors. Within a minute, a running application has a probability of p_s of being suspended, and a suspended application has a probability of p_r of being released again. They are scheduled using the compared approaches. Except for EES, which actively switches unnecessary processors into *sleep* mode and migrates tasks to *active* processors, other approaches schedule DAG applications with a schedule designed offline for all DAG applications, i.e., newly released applications assign and prioritize their tasks based on the previously designed schedule.

First, we investigate the energy consumption of different approaches when handling dynamic applications. We set $p_s = 0.1$ and $p_r = 0.1$



(a) Energy consumption w.r.t., DAG application count

(b) Energy consumption w.r.t., processor count

(c) Energy consumption w.r.t., frequency scalar

Fig. 7. Energy consumption of compared approaches with regard to different parameters.

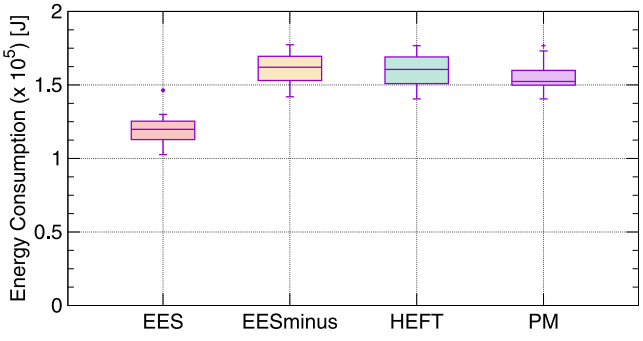


Fig. 8. Energy consumption of compared approaches in an hour.

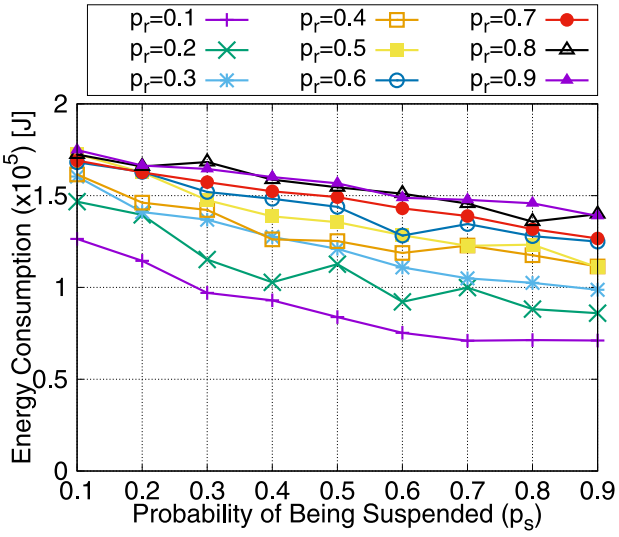


Fig. 9. Energy consumption with respect to different probability of applications being suspended or released.

and perform 20 simulation tests with a simulation time of 60 min. The energy consumption of all approaches is collected and presented in Fig. 8 in a boxplot. Because random assignment is not able to find a feasible schedule with 10 DAG applications on 10 processors, the result of Random is not presented in Fig. 8. We find that EES has the lowest energy consumption, which is around 10%–20% less than the energy consumption of other approaches. Because PM has switched unnecessary processors to *sleep* mode in advance, its energy consumption is slightly less than that of EESminus and HEFT.

Then, we investigate the influence of p_s and p_r on the energy consumption of EES by varying p_s and p_r from 0.1 to 0.9. The results are presented in Fig. 9. It can be found that the energy consumption decreases with increasing p_s and increases with increasing p_r. This is reasonable because a large p_s makes more applications be suspended, leading to the switching of more processors to *sleep* mode, thus reducing energy consumption. On the contrary, when p_r increases, more applications are released at runtime, and more processors should stay on *active* mode to process them. As a result, more energy is consumed.

8. Discussion

In the developed scheduling algorithms, tasks' worst-case execution cycles are taken into account with the aim of strictly meeting constraints. However, tasks' actual execution cycles are often shorter than worst-case execution cycles. It may be more effective to reduce energy consumption if the schedule can be adaptively adjusted according to the actual execution cycles. However, this would also be very challenging because any change towards the designed schedule may violate the constraints or lead to the constraint violations in the future. How to adaptively adjust the schedule to reduce energy consumption by taking the variation of execution cycles into account is an interesting and challenging problem.

In addition, the complexity of the proposed EES and EESminus is $O(m \cdot |\mathcal{T}_{h'}| \cdot \sum_{i=1}^{n} |\mathcal{T}_i|)$ and $O(|\mathcal{T}_{h'}| \cdot \sum_{i=1}^{n} |\mathcal{T}_i|)$, respectively, where m is the number of active processors, and $|\mathcal{T}_{h'}|$ is the number of tasks needing to be migrated out, and $|\mathcal{T}_i|$ is the number of remained tasks. Random, HEFT and PM design their schedules offline, and these schedules would not change at runtime. Thus, their complexity online is $O(1)$. It can be seen that the complexity of our proposed approach is still high. A solution to reduce its complexity is that schedules for handling the suspension of different applications are designed in advance. Whenever an application is suspended, the schedule is adjusted based on these previously designed schedules. In this way, the complexity of the schedule adjustment strategy would become $O(1)$.

9. Conclusion

Intelligent IoT devices drive the embedded system to transform the computing architecture from a traditional single-core to a heterogeneous multiprocessor. The energy consumption problem is becoming more and more severe for many reasons, such as working efficiency and size limits. In this paper, we study how to schedule dynamic applications on a heterogeneous embedded system on the fly. Unlike previous scheduling approaches that are incapable of providing a sufficient guarantee on an application's response time and reliability, we derive an upper bound on an application's response time and reliability. On top of this, we present how to design a schedule for a set of static applications using as few processors as possible. Furthermore, we develop a task migration scheme and a schedule adjustment algorithm, aiming to reduce energy consumption by switching unnecessary processors to *sleep*. The experimental results confirm the superiority of our proposed approaches in terms of schedulability and energy efficiency in comparison to some classic heuristics and a state-of-the-art approach.

In the future, we are going to evaluate the performance of the proposed approaches in real-life embedded systems. In addition, the influence of overhead like task migration and processor mode switching will be investigated.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Funding

This work was supported in part by Science and Technology Innovation 2030 of China under Grant 2021ZD0201401, and in part by National Natural Science Foundation of China under Grant 61802013.

References

[1] G. Hager, J. Treibig, J. Habich, G. Wellein, Exploring performance and power properties of modern multi-core chips via simple machine models, Concurr. Comput.: Pract. Exper. 28 (2012) 189–210.

[2] E. Shamsa, A. Kanduri, A.-M. Rahmani, P. Liljeberg, Energy-performance co-management of mixed-sensitivity workloads on heterogeneous multi-core systems, in: 2021 26th Asia and South Pacific Design Automation Conference, ASP-DAC, 2021, pp. 421–427.

[3] E. Shamsa, A. Kanduri, P. Liljeberg, A.-M. Rahmani, Concurrent application bias scheduling for energy efficiency of heterogeneous multi-core platforms, IEEE Trans. Comput. 71 (2022) 743–755.

[4] B. Hu, Z. Cao, Minimizing resource consumption cost of DAG applications with reliability requirement on heterogeneous processor systems, IEEE Trans. Ind. Inform. 16 (12) (2020) 7437–7447.

[5] A. Paolillo, P. Rodriguez, N. Veshchikov, J. Goossens, B. Rodriguez, Quantifying energy consumption for practical fork-join parallelism on an embedded real-time operating system, in: Proceedings of the 24th International Conference on Real-Time Networks and Systems, 2016, pp. 329–338.

[6] B. Hu, Z. Cao, M. Zhou, Energy-minimized scheduling of real-time parallel workflows on heterogeneous distributed computing systems, IEEE Trans. Serv. Comput. 15 (5) (2021) 2766–2779.

[7] Y. Zhao, H. Zeng, The concept of unschedulability core for optimizing real-time systems with fixed-priority scheduling, IEEE Trans. Comput. 68 (2019) 926–938.

[8] Y. Ma, T. Chantem, R.P. Dick, X.S. Hu, Improving system-level lifetime reliability of multicore soft real-time systems, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 25 (6) (2017) 1895–1905.

[9] S.K. Baruah, Task partitioning upon heterogeneous multiprocessor platforms, in: IEEE Real-Time and Embedded Technology and Applications Symposium, Citeseer, 2004, pp. 536–543.

[10] Z. Wang, P. Li, Z. Liu, K. Wang, W. Xi, H. Yao, X. Jiang, K. Huang, Research on joint optimal scheduling of task energy efficiency and reliability in heterogeneous multiprocessor real-time system, in: IEEE 2nd International Conference on Power, Electronics and Computer Applications, ICPECA, 2022, pp. 17–22.

[11] J. Zhou, T. Wang, W. Jiang, H. Chai, Z. Wu, Decomposed task scheduling for security-critical mobile cyber–physical systems, IEEE Internet Things J. 9 (22) (2021) 22280–22290.

[12] G. Xie, Y. Chen, X. Xiao, C. Xu, R. Li, K. Li, Energy-efficient fault-tolerant scheduling of reliable parallel applications on heterogeneous distributed embedded systems, IEEE Trans. Sustain. Comput. 3 (3) (2017) 167–181.

[13] J. Zhou, K. Cao, X. Zhou, M. Chen, T. Wei, S. Hu, Throughput-conscious energy allocation and reliability-aware task assignment for renewable powered in-situ server systems, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 41 (3) (2021) 516–529.

[14] K. Huang, X. Jiang, X. Zhang, R. Yan, K. Wang, D. Xiong, X. Yan, Energy-efficient fault-tolerant mapping and scheduling on heterogeneous multiprocessor real-time systems, IEEE Access 6 (2018) 57614–57630.

[15] 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 Trans. Reliab. 71 (1) (2020) 178–189.

[16] B. Hu, Y. Shi, Z. Cao, M. Zhou, A hybrid scheduling framework for mixed real-time tasks in an automotive system with vehicular network, IEEE Trans. Cloud Comput. (2022) 1–14, http://dx.doi.org/10.1109/TCC.2022.3194713.

[17] R.I. Davis, A. Burns, A survey of hard real-time scheduling for multiprocessor systems, ACM Comput. Surv. 43 (4) (2011) 1–44.

[18] Y. Wang, K. Li, H. Chen, L. He, K. Li, Energy-aware data allocation and task scheduling on heterogeneous multiprocessor systems with time constraints, IEEE Trans. Emerg. Top. Comput. 2 (2) (2014) 134–148.

[19] Z. Deng, H. Shen, D. Cao, Z. Yan, H. Huang, Task scheduling on heterogeneous multiprocessor systems through coherent data allocation, Concurr. Comput.: Pract. Exper. 33 (10) (2021) e6183.

[20] Z. Deng, D. Cao, H. Shen, Z. Yan, H. Huang, Reliability-aware task scheduling for energy efficiency on heterogeneous multiprocessor systems, J. Supercomput. 77 (10) (2021) 11643–11681.

[21] Z. Cheng, J. Xue, H. Zhang, Z. You, Q. Hu, Y. Lim, Scheduling heterogeneous multiprocessor real-time systems with mixed sets of task, in: 2020 IEEE International Conference on Service Oriented Systems Engineering, SOSE, 2020, pp. 72–81.

[22] J. Qiao, L. Liu, F. Guan, W. Zheng, The BH-mixed scheduling algorithm for DAG tasks with constrained deadlines, J. Syst. Archit. 131 (2022) 102692.

[23] U.U. Tariq, H. Ali, L. Liu, J. Hardy, M. Kazim, W. Ahmed, Energy-aware scheduling of streaming applications on edge-devices in IoT-based healthcare, IEEE Trans. Green Commun. Netw. 5 (2) (2021) 803–815.

[24] J. Zhou, J. Sun, P. Cong, Z. Liu, X. Zhou, T. Wei, S. Hu, Security-critical energy-aware task scheduling for heterogeneous real-time MPSoCs in IoT, IEEE Trans. Serv. Comput. 13 (4) (2020) 745–758.

[25] J. Huang, R. Li, X. Jiao, Y. Jiang, W. Chang, Dynamic DAG scheduling on multiprocessor systems: Reliability, energy, and makespan, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 39 (11) (2020) 3336–3347.

[26] J. Huang, H. Sun, F. Yang, S. Gao, R. Li, Energy optimization for deadline-constrained parallel applications on multi-ECU embedded systems, J. Syst. Archit. 132 (2022) 102739.

[27] J. Huang, R. Li, J. yao An, H. Zeng, W. Chang, A DVFS-weakly dependent energy-efficient scheduling approach for deadline-constrained parallel applications on heterogeneous systems, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 40 (12) (2021) 2481–2494.

[28] M. Han, T. Zhang, Y. Lin, Q. Deng, Federated scheduling for typed DAG tasks scheduling analysis on heterogeneous multi-cores, J. Syst. Archit. 112 (2021) 101870.
[29] L. Zhang, K. Li, C. Li, K. Li, Bi-objective workflow scheduling of the energy consumption and reliability in heterogeneous computing systems, Inform. Sci. 379 (2017) 241–256.
[30] Y. Xu, K. Li, L. He, L. Zhang, K.-C. Li, A hybrid chemical reaction optimization scheme for task scheduling on heterogeneous computing systems, IEEE Trans. Parallel Distrib. Syst. 26 (12) (2015) 3208–3222.
[31] T. Weng, X. Zhou, K. Li, P. Peng, K.-C. Li, Efficient distributed approaches to core maintenance on large dynamic graphs, IEEE Trans. Parallel Distrib. Syst. 33 (1) (2022) 129–143.
[32] S. Tian, W. Ren, Q. Deng, S. Zou, Y. Li, A predictive energy consumption scheduling algorithm for multiprocessor heterogeneous system, IEEE Trans. Green Commun. Netw. 6 (2) (2021) 979–991.
[33] J. Zhou, J. Sun, M. Zhang, Y. Ma, Dependable scheduling for real-time workflows on cyber–physical cloud systems, IEEE Trans. Ind. Inform. 17 (11) (2020) 7820–7829.
[34] B. Hu, S. Xu, Z. Cao, M. Zhou, Safety-guaranteed and development cost-minimized scheduling of DAG functionality in an automotive system, IEEE Trans. Intell. Transp. Syst. 23 (4) (2022) 3074–3086.
[35] B. Hu, Z. Cao, M. Zhou, Scheduling real-time parallel applications in cloud to minimize energy consumption, IEEE Trans. Cloud Comput. 10 (1) (2022) 662–674.
[36] B. Hu, Y. Shi, Z. Cao, Adaptive energy-minimized scheduling of real-time applications in vehicular edge computing, IEEE Trans. Ind. Inform. (2022) 1–11, http://dx.doi.org/10.1109/TII.2022.3207754.
[37] G. Xie, G. Zeng, Z. Li, R. Li, K. Li, Adaptive dynamic scheduling on multi-functional mixed-criticality automotive cyber-physical systems, IEEE Trans. Veh. Technol. 66 (8) (2017) 6676–6692.
[38] G. Xie, L. Liu, L. Yang, R. Li, Scheduling trade-off of dynamic multiple parallel workflows on heterogeneous distributed computing systems, Concurr. Comput.: Pract. Exper. 29 (2) (2017) e3782.
[39] Y. Bai, Y. Huang, G. Xie, R. Li, W. Chang, ASDYS: Dynamic scheduling using active strategies for multifunctional mixed-criticality cyber–physical systems, IEEE Trans. Ind. Inform. 17 (8) (2020) 5175–5184.
[40] G. Xie, H. Peng, X. Xiao, Y. Liu, R. Li, Design flow and methodology for dynamic and static energy-constrained scheduling framework in heterogeneous multicore embedded devices, ACM Trans. Des. Autom. Electron. Syst. 26 (5) (2021) 1–18.
[41] C.Q. Wu, X. Lin, D. Yu, W. Xu, L. Li, End-to-end delay minimization for scientific workflows in clouds under budget constraint, IEEE Trans. Cloud Comput. 3 (2) (2015) 169–181.
[42] Y. Samadi, M. Zbakh, C. Tadonki, E-HEFT: Enhancement heterogeneous earliest finish time algorithm for task scheduling based on load balancing in cloud computing, in: International Conference on High Performance Computing & Simulation, 2018, pp. 601–609.
[43] W. Chen, G. Xie, R. Li, Y. Bai, C. Fan, K. Li, Efficient task scheduling for budget constrained parallel applications on heterogeneous cloud computing systems, Future Gener. Comput. Syst. 74 (2017) 1–11.
[44] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Trans. Dependable Secure Comput. 1 (1) (2004) 11–33.
[45] S.M. Shatz, J.-P. Wang, Models and algorithms for reliability-oriented task-allocation in redundant distributed-computer systems, IEEE Trans. Reliab. 38 (1) (1989) 16–27.
[46] M. Lin, Y. Pan, L.T. Yang, M. Guo, N. Zheng, Scheduling co-design for reliability and energy in cyber-physical systems, IEEE Trans. Emerg. Top. Comput. 1 (2) (2017) 353–365.
[47] L. Zhang, K. Li, Y. Xu, J. Mei, F. Zhang, K. Li, Maximizing reliability with energy conservation for parallel task scheduling in a heterogeneous cluster, Inform. Sci. 319 (2015) 113–131.
[48] L. Zhang, K. Li, K. Li, Y. Xu, Joint optimization of energy efficiency and system reliability for precedence constrained tasks in heterogeneous systems, Int. J. Electr. Power Energy Syst. 78 (2016) 499–512.
[49] S.M. Shatz, J.P. Wang, Models and algorithms for reliability-oriented task-allocation in redundant distributed-computer systems, IEEE Trans. Reliab. 38 (1) (2002) 16–27.
[50] A. Girault, H. Kalla, A novel bicriteria scheduling heuristics providing a guaranteed global system failure rate, IEEE Trans. Dependable Secure Comput. 6 (4) (2009) 241–254.
[51] A. Benoit, L.C. Canon, E. Jeannot, Y. Robert, Reliability of task graph schedules with transient and fail-stop failures: complexity and algorithms, J. Sched. 15 (5) (2012) 615–627.
[52] B. Zhao, H. Aydin, D. Zhu, On maximizing reliability of real-time embedded applications under hard energy constraint, IEEE Trans. Ind. Inform. 6 (3) (2010) 316–328.
[53] B. Zhao, H. Aydin, D. Zhu, Shared recovery for energy efficiency and reliability enhancements in real-time applications with precedence constraints, ACM Trans. Des. Autom. Electron. Syst. 18 (2) (2013) 1–21.
[54] Z. Quan, Z.-J. Wang, T. Ye, S. Guo, Task scheduling for energy consumption constrained parallel applications on heterogeneous computing systems, IEEE Trans. Parallel Distrib. Syst. 31 (5) (2019) 1165–1182.
[55] B. Hu, Z. Cao, L. Zhou, Adaptive real-time scheduling of dynamic multiple-criticality applications on heterogeneous distributed computing systems, in: IEEE 15th International Conference on Automation Science and Engineering, CASE, 2019, pp. 897–903.
[56] M. Park, Non-preemptive fixed priority scheduling of hard real-time periodic tasks, in: International Conference on Computational Science, 2007, pp. 881—888.
[57] H. Topcuoglu, S. Hariri, M.-y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 13 (3) (2002) 260–274.
[58] I. Bate, A. Burns, R.I. Davis, An enhanced bailout protocol for mixed criticality embedded software, IEEE Trans. Softw. Eng. 43 (2017) 298–320.
[59] B. Hu, K. Huang, P. Huang, L. Thiele, A. Knoll, On-the-fly fast overrun budgeting for mixed-criticality systems, in: 2016 International Conference on Embedded Software, EMSOFT, 2016, pp. 1–10.
[60] B. Hu, L. Thiele, P. Huang, K. Huang, C. Griesbeck, A. Knoll, FFOB: efficient online mode-switch procrastination in mixed-criticality systems, Real-Time Syst. 55 (2019) 471–513.
[61] Product specifications, 2023, https://www.mouser.com/pdfdocs/imx7-fact-sheet.pdf. (Accessed 3 February 2023).
[62] R.P. Dick, D.L. Rhodes, W. Wolf, TGFF: task graphs for free, in: IEEE Proceedings of the Sixth International Workshop on Hardware/Software Codesign, CODES/CASHE'98, 1998, pp. 97–101.