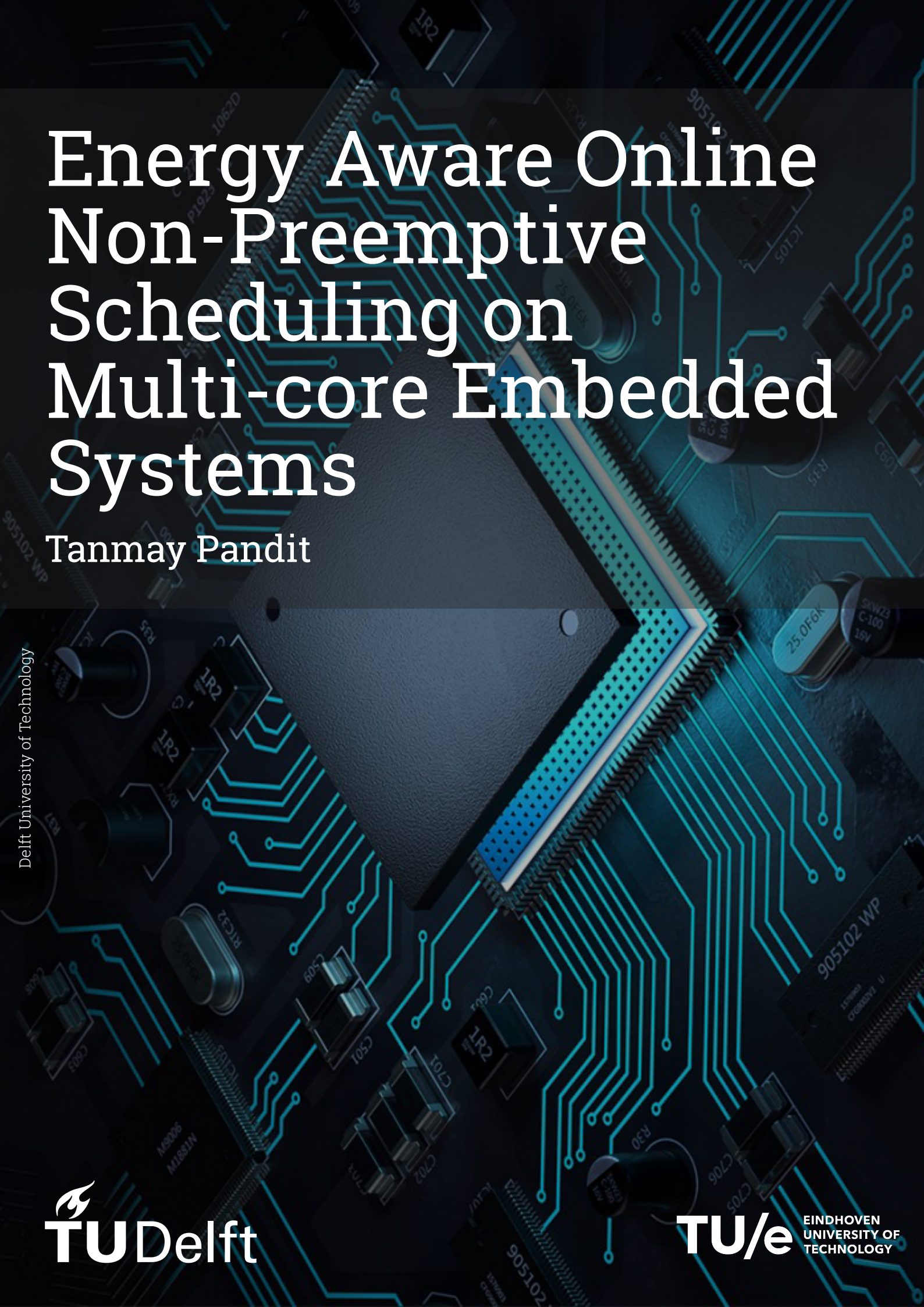
Energy Aware Online Non-Preemptive Scheduling on Multi-core Embedded Systems

Tanmay Pandit



Delft University of Technology

TU Delft

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Energy Aware Online Non-Preemptive Scheduling on Multi-core Embedded Systems

by

Tanmay Pandit

to obtain the degree of Master of Science in
Embedded Systems at the Delft University of Technology,
to be defended publicly on Thursday August 29, 2024 at 09:00 AM.

Project duration:	November 15, 2023 - August 29, 2024	
Thesis committee:	Prof. dr. K. G. Langendoen,	TU Delft
	Dr. M. Nasri,	TU Eindhoven
	Dr. G. Iosifidis,	TU Delft
	P. Gohari-Nazari,	TU Eindhoven

This thesis is confidential and cannot be made public until August 29, 2024.
An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Cover:	IoT Microcontroller Market by World Research Reports (Modified)
Style:	TU Delft Report Style (Modified)

TU Delft

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Summary

Embedded real-time systems, essential in industries like automotive, aviation, and medical devices, increasingly rely on multi-core platforms for efficient parallel processing to meet rising computational demands. However, they face significant challenges with energy consumption, particularly in energy-constrained environments where high energy usage affects system reliability and longevity. Consequently, researchers have focused on enhancing energy efficiency by dynamically adjusting the energy consumption profiles of system components to align with workload demands by leveraging energy management techniques such as dynamic voltage and frequency scaling (DVFS) while meeting timing requirements.

In this work, we consider the problem of energy-aware speed assignment for a hard real-time workload scheduled with an online global non-preemptive work-conserving job-level-fixed-priority scheduler on a multi-core platform with a discrete core-level DVFS model. Our solution determines a speed for each job in the workload to reduce the energy consumption of the system while ensuring the timing requirements of the workload. As we consider a non-preemptive execution model, the timing uncertainties in the workload can result in scheduling anomalies. Therefore, we use the schedule abstraction graph (SAG) [1], a reachability-based response time analysis tool to explore all possible execution scenarios and identify any potential deadline violations.

The key idea of our work is to iteratively explore all execution scenarios using SAG with all jobs running at the most energy-efficient speeds while readjusting the speeds to resolve any potential deadline violation. To limit the speed readjustment search space, we present a novel approach for identifying connections between jobs scheduled on a multi-core platform, considering all possible speed combinations with the selected speed settings.

Our evaluations show that our solution can reduce the energy consumption of a system by 25.85%, on average with an average runtime overhead of 7.7 times higher than that of the schedulability analysis when running all jobs at the highest speed, for the speed range $\{0.74, 0.80, 0.87, 0.94, 1.00\}$. This demonstrates a considerable energy reduction potential with energy-aware speed assignment for static slack reclamation using a schedule abstraction graph.

Contents

Summary i

Nomenclature iv

1 Introduction 1
- 1.1 Problem definition . . . 1
- 1.2 Research goal . . . 3
- 1.3 Challenges and research questions . . . 3
- 1.4 Contributions . . . 4
- 1.5 Thesis organization . . . 5

2 Related work 6
- 2.1 Energy-aware scheduling . . . 6
  - 2.1.1 Single-core energy-aware scheduling . . . 6
  - 2.1.2 Multi-core energy-aware scheduling . . . 7
  - 2.1.3 Summary . . . 9
- 2.2 Schedulability tests . . . 9
  - 2.2.1 Summary . . . 10

3 System model 12
- 3.1 workload and execution model . . . 12
  - 3.1.1 Job set generation from a periodic task set . . . 13
- 3.2 Platform and speed model . . . 14
- 3.3 Power and energy model . . . 15

4 Schedule abstraction graph 16
- 4.1 Schedule abstraction graph . . . 16
  - 4.1.1 Schedule abstraction graph construction . . . 17
- 4.2 Limitations of SAG . . . 21

5 Energy aware scheduling 22
- 5.1 Preprocessing . . . 24
- 5.2 Creating schedule abstraction graph at the lowest speeds . . . 25
- 5.3 Connection with dispatched job . . . 27
  - 5.3.1 Ultimate schedule abstraction graph . . . 30
  - 5.3.2 Causal connection . . . 32
  - 5.3.3 Causal link . . . 35
- 5.4 Energy-aware speed readjustment . . . 40
- 5.5 Timeout . . . 47

6 Empirical evaluation 49
- 6.1 Experimental setup . . . 49
- 6.2 Impact of system utilization . . . 52
- 6.3 Impact of causal link branching heuristic . . . 55
- 6.4 Impact of search space threshold . . . 56
- 6.5 Impact of feasible solution threshold . . . 57
- 6.6 Impact of causal link exploration threshold . . . 58
- 6.7 Impact of the number of tasks . . . 58
- 6.8 Impact of number of cores . . . 60
- 6.9 Impact of execution time variation . . . 61

7 Conclusion **63**
7.1 Summary of contributions 63
7.2 Conclusions 63
7.3 Future work 64

References **66**

Nomenclature

Abbreviations

Abbreviation	Definition
G-NP-EDF	Global Non-preemptive Earliest Deadline First
G-NP-FP	Global Non-preemptive Fixed Priority
JLFP	Job Level Fixed Priority
BCET	Best Case Execution time
WCET	Worst Case Execution time
P-EDF	Partitioned Earliest Deadline First
SAG	Schedule Abstraction Graph
DAG	Directed Acyclic Graph
DVFS	Dynamic Voltage and Frequency Scaling
DPM	Dynamic Power Management

Symbols

Symbol	Definition
$\mathcal{J}$	A set of jobs
$\mathcal{T}$	A set of task
τ_i	A task with index i
t	a time instant
J_i	Job with index i
d_i	The absolute deadline of the job J_i
p_i	The priority of the job J_i
r_i^{min} , r_i^{max}	Earliest and latest arrival time of a job J_i
C_i^{min} , C_i^{max}	best case and worst case execution time of a job J_i
m	Number of cores
m_i	core with index i
$\mathcal{E}_{\mathcal{S}}$	Energy consumption at speed $\mathcal{S}$
$\mathcal{E}_{i,\mathcal{S}_k}$	Energy consumption of job J_i at speed $\mathcal{S}_k$
V_{dd}	Supply voltage
C_{ef}	Effective switching capacitance
$\mathcal{S}$	Speed
$\mathcal{S}^{crtl}$	critical Speed
f	Core frequency
f^{min} , f^{max}	Minimum and maximum frequency
E	Set of edges
V	Set of vertices
P	Path
A	Set of system availability intervals
$R_{analysis}$	Runtime of the analysis
$\mathcal{E}_{analysis}$	Energy consumption with speed assignment of the analysis

1

Introduction

Embedded real-time systems is pivotal across numerous industries, including automotive, aviation, medical, industrial automation, and consumer electronics. These systems are often safety-critical, namely, a wrong action or an action at the wrong time can result in human loss or damage to the environment. Therefore, these systems need both functional and temporal correctness. For instance, in the automotive sector, real-time requirements need to be met to ensure that the anti-lock braking systems (ABS) and airbags work according to the timing requirements of the safety standards [2].

The demand for increased computational power in embedded systems has led to the widespread adoption of multi-core platforms [3]. Multi-core platforms facilitate parallelism, enabling the handling of more complex tasks concurrently [3]. Although these emerging architectures provide noticeable performance benefits, coordinating real-time tasks and shared resources across multiple cores while meeting strict timing constraints presents significant challenges [4]. Additionally, they also demand a significant amount of energy to support the increasing number of cores [4].

1.1. Problem definition

While predictability, correctness, and performance are essential characteristics of real-time embedded systems, energy consumption is also a crucial design consideration, as it impacts battery life, reliability, and total cost of the system [5]. Especially in energy-constraint embedded systems, the need for energy-efficient systems is significant where increased energy consumption can harm the system's longevity, compromising the system's reliability in critical applications. High energy consumption can also result in increased system heat dissipation, reducing the system's functional correctness (reliability) with increased system failure [5].

With efforts to reduce energy consumption at the hardware level with low-power embedded systems, many researchers [6]–[11] have also focused on

energy-aware scheduling to improve energy efficiency at the system level by actively changing the power consumption profile of system components to effectively match workload demands while ensuring the timing requirements of the real-time systems. On modern embedded platforms, two widely employed techniques for this purpose are Dynamic Power Management (DPM) and Dynamic Voltage and Frequency Scaling (DVFS).

DPM focuses on reducing energy consumption by entering the processor into a low-power inactive state during periods of inactivity. This technique primarily addresses static power consumption, which is the power dissipation due to leakage currents that persist as long as the system is powered on, regardless of the system activity level. By maximizing the time the processor spends in this low-power state, DPM can significantly reduce overall energy usage, while ensuring that all real-time tasks are completed within their deadlines.

On the other hand, DVFS operates by reducing the voltage and frequency of the processor to effectively lower the dynamic power consumption, which is the power associated with the active operation of the processor. However, a reduction in frequency also leads to an increase in task execution times. Therefore, it is crucial to determine the optimal processor speeds that balance energy savings while ensuring that all tasks meet their timing constraints. In an ideal scenario, DVFS processors would offer an unlimited, continuous range of voltages and frequencies, a precisely defined power-frequency relationship, and no overhead in switching speeds. However, the practical DVFS implementations are constrained due to the significant cost, complexity, and impracticality of an ideal DVFS [12]. This non-ideal DVFS provides a limited selection of discrete voltages and frequencies and incurs overhead associated with processor speed switching.

In energy-aware scheduling, the scheduler uses DVFS to reduce energy consumption by utilizing the slack time, i.e., the time when the processor is idle in a schedule. The scheduler extends task execution into this slack time by lowering processor frequency with DVFS, thereby reducing energy consumption. This technique is known as slack reclamation [11]. Static slack reclamation approaches are design time solutions that consider the slack time based on worst-case execution time (WCET) analysis. In practice, the actual execution time (AET) is less than WCET. For a schedule based on WCET, the difference between AET and WCET provides additional slack time that is unclaimed by static slack reclamation. To reclaim the runtime slack, an online slack reclamation technique known as dynamic slack reclamation [11] is used.

Energy-aware scheduling on multi-core platforms with DVFS has been studied extensively for a preemptive execution model. In this model, an executing job can be preempted by another higher-priority job to maintain schedulability (the ability of the system to respect all deadlines under a given scheduling policy). In contrast, the non-preemptive execution model which does not allow preempting of jobs while executing, has received limited attention [11], [13]. However, the non-preemptive execution model remains appealing in various applications where the characteristics of device hardware and

software make preemption prohibitively expensive or impossible [14]. Compared to preemptive algorithms, non-preemptive scheduling algorithms are easier to implement, have lower runtime overhead, and require less memory [14]. Non-preemptive scheduling also makes programs more suitable for worst-case execution time analysis, as it preserves the program locality [8], [15]. As a result, non-preemptive scheduling has been widely adopted in many avionics applications and embedded systems, particularly in small embedded devices with limited memory capacity [16].

The previous works for energy-aware scheduling with the non-preemptive execution model on a multi-core platform that ensures schedulability, such as those discussed by Bambagini et al. [11], mainly focus on an offline scheduling approach or a partitioned scheduling [17]. Although offline schedule can provide optimal solution [11], it is often impractical for embedded platforms due to memory constraints. This is because a table-based offline scheduling solution requires considerable memory resources to store the schedule [18]. Similarly, the partitioned multi-core scheduling approach considers a multi-core scheduling problem as a multiple single-core scheduling problem by assigning each task to a core. While partitioned scheduling is highly studied, the effectiveness of partitioned scheduling is limited by the partitioning or bin-packing problem, which is a known NP-hard problem [19]. Consequently, global scheduling [17] where a job can be assigned to any core, has easier and more efficient load balancing across cores compared to a partitioned scheduling approach as job assignment to the core is not limited by sub-optimal partitioning heuristic [20].

1.2. Research goal

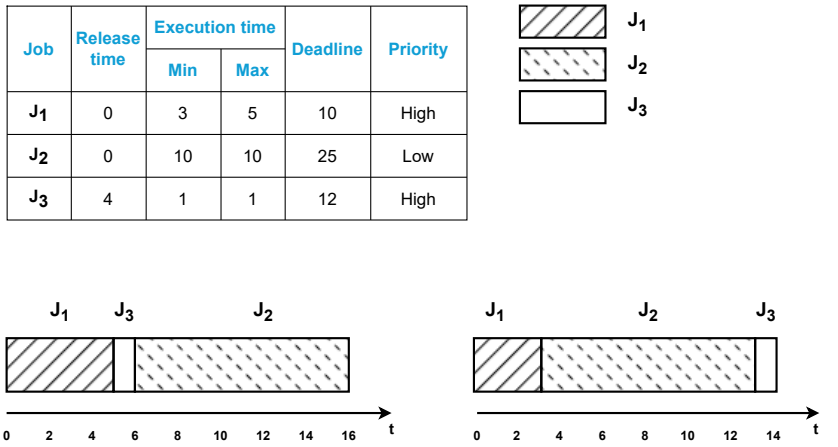
To the best of our knowledge, there has been no prior research addressing energy-aware online global hard real-time scheduling on multi-core platforms using a non-preemptive execution model and discrete DVFS. This thesis aims to introduce a static slack reclamation method for assigning an energy-efficient speed to each job [1] in a workload scheduled with an online global non-preemptive work-conserving job-level fixed-priority(JLFP) [21] scheduler on a multi-core platform with a discrete DVFS model, where each core can execute at one of the predefined set of speeds. The objective is to maintain the schedulability of the system while reducing energy consumption.

1.3. Challenges and research questions

While the non-preemptive execution model has lower runtime overhead and more predictability than the preemptive execution model, the schedulability analysis for the non-preemptive execution model with workload timing uncertainties is difficult. As jobs cannot be preempted once execution is started, the runtime uncertainties in the execution time or release time of a job can result in scheduling anomalies that cannot be identified by utilization-based schedulability tests.

[1] A job is an instance of a task in a workload.

Job	Release time	Execution time		Deadline	Priority
		Min	Max		
J_1	0	3	5	10	High
J_2	0	10	10	25	Low
J_3	4	1	1	12	High



(a) Scenario 1: J_1 finishes execution at time 5 (b) Scenario 2: J_1 finishes execution at time 3

Figure 1.1: Scheduling anomalies in non-preemptive scheduling

Example 1. Consider an example where three jobs are scheduled on a single-core platform with a non-preemptive job-level fixed-priority scheduling policy. Job J_1 starts the execution at time 0 as it has the highest priority amongst ready jobs. As J_1 has execution time uncertainty, it can finish its execution as early as 3 and as late as 5. Figure 1.1(a) considers the scenario when J_1 finishes the execution at 5. At time 5, J_3 is the highest priority job amongst all the ready jobs. Therefore, J_3 will start its execution and all jobs are scheduled successfully. Figure 1.1(b) considers the scenario when J_1 finishes the execution at 3. As J_2 is the only ready job at time 3, J_2 starts its execution. Due to the non-preemptive execution model, higher priority job J_3 has to wait for J_2 to finish its execution. This results in a deadline miss for J_3.

The schedulability analysis of a non-preemptive workload with timing uncertainties is further complicated when scheduled on a multiprocessor with a global scheduling policy where ready jobs can be assigned to any of the available cores. The execution time variation results in core availability time uncertainty. In addition to the multiple possible ordering anomalies discussed in example 1, the core availability time uncertainty can result in multiple possible core assignment anomalies.

This leads to our main research question.

RQ 1. How can jobs be scheduled at specific speeds with an online non-preemptive global scheduling policy on a multi-core platform with discrete speeds to reduce the energy consumption of a system, while ensuring that jobs meet their deadline requirements?

1.4. Contributions

We present the first work in energy-aware online global hard real-time scheduling with a non-preemptive job-level fixed-priority (JLFP) scheduling

policy on a homogeneous multi-core platform with an arbitrary number of cores and discrete speed settings. We propose a static slack reclamation approach to assign an energy-aware speed to each job in the job set to reduce energy consumption while maintaining the schedulability.

1.5. Thesis organization

The thesis is structured in six sections. Chapter 2 provides an overview of previous works related to energy-aware real-time scheduling and schedulability analysis for global non-preemptive scheduling. Chapter 3 describes the system, workload, and energy models used in this work that defines the scope of our work. Based on the existing work in schedulability analysis for the global non-preemptive execution model, chapter 4 discusses the schedulability and response time analysis framework used in the proposed solution. Chapter 5 presents an approach for energy-aware scheduling of non-preemptive workload on a multi-core platform with discrete speed DVFS. We further evaluate the presented approach for multi-core energy-aware scheduling in chapter 6. Finally, we conclude the work in chapter 7 with a discussion on the presented approach and insights into future improvements.

2

Related work

This chapter reviews the related work on energy-aware scheduling and schedulability analysis for homogeneous multi-core platforms using a global non-preemptive execution model.

2.1. Energy-aware scheduling

2.1.1. Single-core energy-aware scheduling

Energy-aware scheduling has been extensively studied on single-core platforms with both online and offline approaches for preemptive and non-preemptive execution models. Since the scope of this thesis is energy-aware scheduling on multi-core platforms with a non-preemptive execution model, the related work for preemptive single-core scheduling is not discussed.

Zhang and Chanson [6] proposed the dual speed (DS) algorithm based on the EDF policy for a single-core platform. This work considers periodic preemptive jobs with non-preemptive critical sections (blocking sections). A scenario where a high-priority job cannot be executed because a lower-priority job is executing a critical section is defined as a blocking scenario. The dual speed (DS) algorithm calculates two speeds: a low speed is determined by assuming the task set is independent, whereas a high speed is calculated by considering the blocking time caused by non-preemption of critical sections. The algorithm runs each task at low speed most of the time, switching to high speed only when a task is blocked. Lee et al. [22] extended this work with a multi-speed algorithm that utilizes various speed levels based on specific blocking situations to minimize energy consumption.

Jejurikar et al. [7] proposed the stack-based slowdown algorithm, which minimizes transitions to a higher speed by computing different slowdown factors based on the blocking task, building on the optimal feasibility test for non-preemptive systems. Li et al. [8] introduced the individual speed algorithm (ISA), which computes a unique speed for each task in a non-preemptive set,

avoiding unnecessary preemption time overestimation and ensuring deadlines are met.

The global scheduling model with a work-conserving policy assigns a ready job to any of the cores of a multi-core platform. As the before-mentioned solutions [6]–[8], [22] do not consider the impact of global scheduling on a multi-core platform, these solutions cannot be used for a system with a global scheduler.

2.1.2. Multi-core energy-aware scheduling

Similar to single-core scheduling, multi-core energy-aware scheduling has received significant attention due to the industry shift to multi-core platforms. Still, it mainly focuses on the preemptive execution model. The work in the non-preemptive execution model for tasks with explicit deadlines is limited to offline or partitioned scheduling.

Homogeneous multi-core platforms

For preemptive scheduling on the homogeneous multi-core platform, Funaoka et al. [23] proposes an optimal static algorithm for continuous frequency scaling in multiprocessors based on the Largest Nodal Remaining Execution time First (LNREF) [20] global scheduling algorithm. LNREF is an optimal real-time scheduling algorithm for multiprocessors that ensures any periodic task set with a utilization less than or equal to the number of cores will be scheduled to meet all deadlines. The solution proposed by Funaoka et al. [23] categorizes tasks into heavy and light tasks based on their utilization. Each heavy task is assigned to a dedicated core while the light tasks are scheduled on the rest of the cores. All cores assigned for the light tasks follow uniform frequency assignments where all cores are allotted the same speed.

Unlike Funaoka et al. [23], the Growing Minimum Frequency (GMF) algorithm [9] assigns optimal frequency to each core on a multiprocessor scheduled with U-LLREF [24] optimal multiprocessor scheduling algorithm designed for homogeneous multiprocessors. GMF leverages the incremental testing approach of U-LLREF, which checks the schedulability of subsets of tasks on subsets of processors by expanding these subsets one element at a time. GMF adjusts the frequency of the processors tested by U-LLREF to accommodate the current subset of tasks effectively and assigns different speeds to each core from discrete speed ranges with uniform speed steps.

El Sayed et al. [25] presented a blocking-aware-based partitioning (BABP) algorithm that addresses the problem of energy-aware static partitioning of periodic, dependent real-time tasks to schedule with Partitioned Earliest-Deadline-First (P-EDF) [17]. BABP assigns tasks with the same shared resources to the same core to ensure the parallel tasks do not share resources. Moreover, the BABP algorithm uses the DS algorithm [6] to execute tasks at a low speed and switch to high speed in a blocking scenario as discussed in subsection 2.1.1.

For non-preemptive scheduling on a homogeneous platform, Zhu et al. [26] presented Global Scheduling with Shared Slack Reclamation (GSSR) al-

gorithm, an energy-aware scheduling approach for a global multi-core platform with a non-preemptive execution model for a frame-based task set. In a frame-based task set, all tasks in the frame have a common deadline. GSSR computes the minimum speed used to execute the selected task without missing the deadline for the current frame while considering the slack provided by the already executed jobs and maintaining a polynomial time complexity.

Shieh et al. [27] proposes an offline and online approach for energy-aware non-preemptive scheduling on a dual-core platform with two speeds. For the offline approach, the authors present an integer linear programming (ILP) formulation that decides the core and voltage assignment for each task. In the online approach, authors present a heuristic that selects the task with the earliest deadline and selects speed and core based on the workload requirement of the ready job and core utilization.

Shieh et al. [10] extended the previous work by considering a system with a homogeneous multi-core platform and discrete speed model for scheduling an aperiodic non-preemptive task set. The authors present an offline approach with an integer linear programming formulation that decides the core and voltage assignment for each task. Additionally, they propose an online heuristic for task-to-core mapping and voltage assignment for each ready task. In their online approach, whenever a core finishes executing a scheduled task, the system calculates the dynamic priority of each ready task based on the ratio of the task's WCET to the remaining time until its deadline. The dynamic priority is used to determine the task assignment for the available core and required voltage for the selected task. Additionally, voltage assignment also considers the timing requirements of all waiting jobs to ensure scheduling feasibility.

The previously mentioned works in preemptive global scheduling [9], [23] are based on scheduling algorithms LNREF and U-LLREF, which are not optimal for non-preemptive scheduling. Although the work presented by Zhu et al. [26] is close to our system model, GSSR considers a frame-based task set where all tasks have a common deadline. Hence, this approach cannot be directly extended to a periodic job set where each task has an explicit deadline. Finally, the approach by Shieh et al. [27] is limited to two cores and two speeds which cannot be applied to a multi-core platform with an arbitrary number of cores and discrete speeds. The proposed solution for energy-aware scheduling on a multi-core platform by Shieh et al. [10] is closest to our system model as a job set from a periodic task set can be similar to an aperiodic task set if each job in the job set is considered as an aperiodic task. Although authors [10] provide a dynamic priority core assignment, no scheduling guarantee or analysis (with or without the voltage assignment) is provided for the proposed heuristic. As the goal of this thesis is to maintain schedulability for a schedulable job set while reducing energy consumption, the proposed heuristic cannot be used for this thesis goal since it does not consider the impact of speed changes on tasks that are not in waiting task.

Heterogeneous multi-core platforms

Guo et al. [28] presented an approach for energy-aware scheduling of sporadic parallel jobs on a clustered multi-core platform with a preemptive execution model where the cores in the cluster are assigned the same frequency. To partition the tasks on the cores, they proposed the concept of the speed-profile to present the energy-consumption behavior for each task and cluster. Similar to Guo et al. [28], Chen et al. [29] discusses the energy-aware offline scheduling problem of dependent tasks upon a heterogeneous multiprocessor platform and presents a list-based scheduling approach. The presented LESA algorithm assigns task priority considering their dependency. Then, it defines an energy-based weight for each task based on total available energy and allocates the task to a processor with a defined speed, start time, and finish time.

For non-preemptive energy-aware scheduling on a heterogeneous platform, Chniter et al. [30] proposes an integer linear programming (ILP) approach to schedule re-configurable periodic tasks on heterogeneous cores by partitioning the task to match the energy consumption to the system's battery capacity. If the partitioning fails, task periods are adjusted and tasks are remigrated to feasibly schedule the jobs under given energy constraints. All discussed solutions [28]–[30] present a partitioned scheduling approach that considers a multi-core scheduling problem as multiple single-core scheduling problems with tasks assigned to a specific core. This approach cannot be applied to our work as it considers the global scheduling model.

2.1.3. Summary

Although energy-aware scheduling has been studied for various system models with different workloads and real-time constraints, the work for energy-aware online scheduling with explicit deadlines on a multi-core platform with a global non-preemptive execution model is missing. Additionally, the approaches presented in the related works do provide a unique perspective to solve the energy-aware scheduling for the respective system models. Still, none of the work can be directly applied to global multi-core hard real-time scheduling with an arbitrary number of cores and discrete speed range with a non-preemptive execution model and explicit deadline workload. The related work is summarized in Table 2.1

2.2. Schedulability tests

Non-preemptive scheduling has received much less attention in research compared to preemptive scheduling. However, it is widely used as a low-overhead solution with high predictability [14]. These benefits are especially significant on multiprocessor platforms, where task migration overhead is higher and more unpredictable. Non-preemptive scheduling provides more predictability in multiprocessor worst-case execution time analysis due to reduced migrations and a higher level of isolation [15]. To the best of our knowledge, there exists no optimal scheduling policy for global non-preemptive multi-core scheduling. Therefore, work-conserving heuristics such as Global Non-Preemptive EDF and Global Non-Preemptive Fixed-Priority [21] are commonly used scheduling policies for global non-

preemptive scheduling. As the objective of our work is to reduce energy consumption while ensuring the timing constraints, it is important to provide timing guarantees for the system scheduled with the selected energy-aware speeds. Therefore, this section will discuss the existing schedulability analysis for global non-preemptive scheduling.

Baruah et al. [31] introduced a sufficient but not necessary polynomial-time schedulability test for global non-preemptive scheduling using the G-NP-EDF algorithm for periodic task sets. This test accounts for the additional interference time resulting from non-preemption. However, a key limitation of this test is that it cannot guarantee the schedulability of a task set if any task has an execution time longer than the shortest relative deadline among the tasks. This constraint applies to a single-core non-preemptive platform, where long execution times can block other tasks from meeting their deadlines. For multi-core scheduling, this constraint does not necessarily apply, as tasks can be scheduled on different processors, allowing them to meet their deadlines even if some cores are occupied for long periods.

Guan et al. [21], [32] presented three schedulability tests for global non-preemptive fixed-priority (G-NP-FP) scheduling. By leveraging the ”problem window analysis” previously used for preemptive multiprocessor scheduling [33], they first developed a linear-time general schedulability test that applies to G-NP-FP as well as any work-conserving non-preemptive global scheduling policy. Further, they proposed pseudo-polynomial time-complexity schedulability test conditions for G-NP-FP and G-NP-EDF scheduling, which provided significant performance enhancements over the initial linear-time test.

Lee et al. [34] focuses on refining response-time bounds for global non-preemptive multi-core scheduling to minimize unnecessary interference from carry-in job sets. They define carry-in jobs as those released before a specific time t and still executing at t. By leveraging these bounds, [34] propose a tighter schedulability test for global non-preemptive fixed-priority (NP-FP) scheduling. Their approach utilizes necessary conditions and takes advantage of reduced interference caused by carry-in jobs.

Recently, Nasri et al. [1], [15] presented a reachability-based response time analysis framework called a schedule abstraction graph (SAG). SAG generates a graph based on all possible scheduling decisions for a global non-preemptive work-conserving scheduling policy. For each state in the graph, it considers the impact of the scheduling decision and explores all possible scheduling scenarios for a given job set to verify the schedulability while providing response time bounds.

2.2.1. Summary

In summary, while Baruah et al. [31], Guan et al. [21], [32], and Lee et al. [34] have primarily focused on providing sufficient schedulability tests tailored for sporadic task sets, these tests tend to become overly pessimistic when applied to periodic tasks because they fail to discount many execution scenarios with sporadic task set that are not present with periodic task set. Nasri et al. ’s SAG framework [1], [15], although not an exact test, explores

all possible scheduling scenarios and provides a tighter schedulability test and a response-time analysis. SAG not only outperforms the other techniques but also provides results within reasonable computation times. Unlike [21], [31], [32], [34], SAG also considers the release jitter that may arise due to timer inaccuracy, interrupt latency, or networking delays. Finally, by exploring all possible scheduling scenarios, information from SAG can effectively guide the assignment of speeds to jobs for energy-aware scheduling.

Table 2.1: The summary of related work

Reference	Task scheduling	Platform	Task set	Global scheduling	Preemption model	Speed model	Schedulability guarantee
[6]	Online	Single-core	Periodic	-	Non-preemptive sections	Two speed discrete	Yes
[22]	Online	Single-core	Periodic	-	Non-preemptive sections	Discrete	Yes
[7]	Online	Single-core	Periodic	-	Non-preemptive	Discrete	Yes
[8]	Online	Single-core	Periodic	-	Non-preemptive	Discrete	Yes
[23]	Online	Homogeneous multi-core	Periodic	Yes	Preemptive	Static continuous/discrete	Yes
[9]	Online	Homogeneous multi-core	Periodic	Yes	Preemptive	Static discrete	Yes
[25]	Online	Homogeneous multi-core	Periodic	No	Preemptive	Two speed discrete	Yes
[26]	Online	Homogeneous multi-core	Frame-based	Yes	Non-preemptive	Continuous / discrete	Yes
[27]	Offline/Online	Homogeneous dual-core	Aperiodic	Yes	Non-preemptive	Two speed discrete	No
[10]	Offline/Online	Homogeneous multi-core	Aperiodic	Yes	Non-preemptive	discrete	No
[28]	Online	Heterogeneous multi-core	Sporadic parallel	No	Preemptive	Continuous / discrete	Yes
[29]	Offline	Heterogeneous multi-core	Dependent	No	Preemptive	Discrete	Yes
[30]	Offline	Heterogeneous multi-core	Re-configurable periodic	No	Non-preemptive	Discrete	Yes
Our work	Online	Homogeneous multi-core	Periodic	Yes	Non-preemptive	Discrete	Yes

3

System model

We address the problem of energy-aware scheduling of a finite set of non-preemptive jobs $\mathcal{J}$ with hard real-time requirements on multi-core platforms with identical cores. This chapter will discuss system models for the selected problem.

3.1. workload and execution model

Workload Model: We define each job $J_i \in \mathcal{J}$ as

$$J_i = ([r_i^{min}, r_i^{max}], [C_i^{min}, C_i^{max}], d_i, p_i), \tag{3.1}$$

with the earliest release time r_i^{min}, latest release time r_i^{max}, best case execution time at the highest speed (BCET) C_i^{min}, worst-case execution time (WCET) at the highest speed C_i^{max}, absolute deadline d_i, and job priority p_i. We consider a discrete-time model where the job timing parameters are integer multiples of the system clock. At runtime, each job may be released at a time r_i within the interval $[r_i^{min}, r_i^{max}]$, where at time instance t, a job J_i is considered possibly released if $t \geq r_i^{min}$ and certainly released if $t \geq r_i^{max}$. This release jitter considers scenarios such as timer inaccuracies, interrupt latency, and communication delays, which can result in variations in job release time [15].

The execution time variation of a job with execution time C_i within the interval $[C_i^{min}, C_i^{max}]$ results from factors such as caching, out-of-order execution, input dependencies, and program path diversity [15]. Additionally, shared resources requiring mutual exclusion are protected by FIFO spin locks [35]. Since we consider a non-preemptive execution model, the worst-case spinning delay while waiting to acquire a lock is considered in the execution time variation.

A job's absolute deadline d_i is fixed and independent of release jitter. The job priority p_i is decided based on the selected job-level fixed priority (JLFP) algorithm or designer-assigned priority. We assume that the numerically

lower value for p_i implies higher priority. Ties in priority are resolved arbitrarily and consistently. For ease of notation, we assume that the "<" operator implicitly reflects this tie-breaking rule.

Scheduling Model: Each job must execute sequentially on a core, meaning it cannot be split into smaller sections to run on multiple cores concurrently. Thus, a job starting its execution on a core at time t occupies that core until time $t + C_i$ and the core becomes available for other jobs at $t + C_i$.

A job is considered ready at time t if it has been released and has not yet started execution before t. Jobs remain pending until completed, with no job-discarding policy. We consider a work-conserving non-preemptive global JLFP scheduler such as G-NP-EDF [21] or G-NP-FP [21] operating on a homogeneous multi-core platform. This scheduler is activated whenever a job is released or completed. The highest priority ready job is selected and assigned to any available core at each invocation. We consider a non-deterministic core-selection policy when more than one core is available or executing a job, i.e., when a job is scheduled, it may be scheduled on any available core. The scheduler does not leave a core idle if a ready job exists. As the JLFP scheduler is deterministic, it always produces the same schedule for a given execution scenario, where an execution scenario is defined as follows (according to [1], [15], [36]).

Definition 1. An execution scenario $\gamma = \{(r_1, C_1), (r_2, C_2), \ldots, (r_n, C_n)\}$, where $n = |\mathcal{J}|$, is an assignment of release times and execution times to the jobs of $\mathcal{J}$ such that, $\forall J_i \in \mathcal{J}$, $r_i \in [r_i^{min}, r_i^{max}]$ and $C_i \in [C_i^{min}, C_i^{max}]$.

As we consider a hard real-time system, a job set $\mathcal{J}$ is schedulable under a given JLFP scheduling policy only if no execution scenario of $\mathcal{J}$ results in a deadline miss.

3.1.1. Job set generation from a periodic task set

Although the presented workload considers a job set, it supports commonly used workload models such as periodic task sets. This section discusses the approach for creating the job set from a periodic task set.

Job set formation: Consider a periodic task set $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$, where each task τ_i is defined as

$$\tau_i = (T_i, D_i, [C_i^{\min}, C_i^{\max}], \epsilon_i),$$

with task period T_i, relative deadline D_i, execution time interval $[C_i^{\min}, C_i^{\max}]$, and release time jitter ϵ_i . The task set $\mathcal{T}$ can be converted into $\mathcal{J}$ as defined in eq. (3.1) as follows:

1. Compute the hyperperiod $H = \text{lcm}(T_1, \ldots, T_n)$.
2. For each task τ_i, calculate the number of jobs it releases within one hyperperiod: $n_i = \lceil \frac{H}{T_i} \rceil$.
3. Generate the job set, $\mathcal{J}$ by including all jobs from all tasks within this hyperperiod as

$$\mathcal{J} = \{J_{i,k} \mid \forall i, 1 \leq i \leq n; \forall k, 1 \leq k \leq n_i\}.$$

Job parameter assignment: After generating each job in the job set, the timing parameters for each job (according to eq. (3.1)) can be assigned. The kth job of τ_i is defined as:

$$J_{i,k} = \left([r_{i,k}^{\min}, r_{i,k}^{\max}], [C_i^{\min}, C_i^{\max}], d_{i,k}, p_{i,k}\right).$$

The release interval of the job is defined as

$$r_{i,k}^{\min} = (k-1)T_i,$$

$$r_{i,k}^{\max} = (k-1)T_i + \epsilon_i,$$

and the absolute deadline of the job is $d_{i,k} = r_{i,k}^{\min} + D_i$.

Job priority assignment: The priority $p_{i,k}$ for job $J_{i,k}$ is assigned according to the selected scheduling policy or designer-assigned priority. For example, the earliest deadline first (EDF) scheduling policy assigns priority for each job based on the absolute deadline of a job[1]. The job with an earlier (numerically smaller) absolute deadline will have a higher priority. As we assume the numerically lower value of p_i implies higher priority, job priority assignment, p_i set to the absolute deadline of the job ensures priority resulting from EDF policy.

Similarly, priority assignment with the rate monotonic (RM) scheduling policy depends on the period of the task where a task with a smaller period will be assigned a higher priority. Therefore, all jobs of the same task will be assigned the same priority based on the period of the task.

3.2. Platform and speed model

We consider a homogeneous multi-core platform composed of m identical processors represented by $\pi = \{\pi_1, \pi_2, \ldots, \pi_m\}$ and job-level dynamic voltage and frequency scaling (DVFS) with discrete speed range. Hence, each core can scale with different DVFS settings independent of other cores based on the assigned speed of the job to be dispatched on that core.

We define operating speed $\mathcal{S}$ as a normalized operating frequency. The normalized operating frequency, $\mathcal{S}_i$ is the ratio of the current core frequency to the maximum processor frequency f^i/f^{max}, where the core is running with frequency f^i. The maximum core speed or the core speed at highest f^{max} is defined as $\mathcal{S}^{max} = 1.0$. Hence, the speed range has discrete values within {$\mathcal{S}^{min}$,$\mathcal{S}^{max}$} where $\mathcal{S}^{min}$ and $\mathcal{S}^{max}$ are the minimum and maximum speeds corresponding to f^{min} and f^{max}, respectively.

We also consider job-specific critical speed $\mathcal{S}^{crtl}$, which, as defined in [37], is the operating speed that minimizes the energy consumption per cycle, i.e., any lower speed results in higher energy consumption than energy consumption at the critical speed. Hence the active operating speed range is within $\{\mathcal{S}^{crtl}, \ldots, \mathcal{S}^{max}\}$.

As observed in [38], the energy and time overhead of DVFS for common embedded multiprocessors are negligible. Therefore, we do not consider

[1] EDF is a task-level dynamic priority policy but a job-level fixed-priority policy.

them as a part of our scheduling model, but instead, we assume they have been included in the value of the WCET of each job, given that the core speed scaling only occurs before the start of execution of a job and its timing overhead is predictable. The relation between the selected speed of the core and the execution time of a job is defined by Equation 3.2.

$$C_i(\mathcal{S}) = \frac{C_i}{\mathcal{S}} \tag{3.2}$$

3.3. Power and energy model

As we assume a platform that can assign the frequency and voltage to each core individually, power consumption for each core in active state is characterized by P which consists of static power consumption and dynamic power consumption as shown in Equation 3.3.

$$P = P_d + P_s \tag{3.3}$$

Dynamic power consumption P_d is given by Equation 3.4, where C_{ef} is the effective switching capacitance, V_{dd_i} is the supply voltage, and f_i is the core frequency [39], [40].

$$P_d = C_{ef}.{V_{dd_i}}^2.f_i \tag{3.4}$$

The core frequency f is dependent on the supply voltage V_{dd} as V_{dd} limits the highest frequency[11]. The frequency-voltage relation is given by Equation 3.5 [41], where k is hardware-design-specific constant and V_t is the threshold voltage. Therefore, V_{dd_i} is the required supply voltage for the frequency f_i

$$f = k\frac{(V_{dd} - V_t)^2}{V_{dd}} \tag{3.5}$$

Similarly, the static power consumption of an active core is given by eq. (3.6), where, V_{dd_i} is the selected core supply voltage for frequency f_i and α_1 and α_2 are hardware-design-specific constants that are independent of active application-specific parameters [38].

$$P_s = \alpha_1.V_{dd_i} + \alpha_2 \tag{3.6}$$

Hence the power consumption at speed $\mathcal{S}_i$ is given by

$$P(\mathcal{S}_i) = C_{ef}.{V_{dd_i}}^2.f_i + \alpha_1.V_{dd_i} + \alpha_2, \tag{3.7}$$

where f_i is the core frequency required for speed $\mathcal{S}_i$. The energy consumed for the core at speed S for period τ is given by $E_{\mathcal{S}} = P(\mathcal{S}) \times \tau$. Based on the Equation 3.2, The energy consumption for a job J_i, $E_{i,\mathcal{S}}$ calculated as

$$E_{i,\mathcal{S}} = P(\mathcal{S}) \times C_i(\mathcal{S}). \tag{3.8}$$

4

Schedule abstraction graph

The approach in this work uses a Schedule abstraction graph (SAG) to explore the different execution scenarios for a given job set and check the schedulability of the job set with analyzed energy-efficient speeds. This chapter introduces SAG, the reachability-based response-time analysis by Nasri et al. [1], [15], [36] and describes how the SAG works.

4.1. Schedule abstraction graph

The SAG searches the space of possible decisions a scheduler can take with a job-level-fixed-priority (JLFP) scheduling policy for a set of jobs $\mathcal{J}$ by building a schedule-abstraction graph (SAG). SAG is a directed acyclic graph (DAG) denoted by $G = (V, E)$ and consists as follows:

Vertices

Each vertex v in the set of vertices V represents the system's state after executing a set of jobs. Each state $v \in V$ for multi-core SAG analysis maintains a set of system-availability intervals (defined in [1]), denoted $A = A_1, A_2, \ldots, A_m$, where $A_x = [A_x^{min}, A_x^{max}]$ means that x cores can be possibly available starting at time A_x^{min} and certainly available no later than at time A_x^{max}. These system-availability intervals represent the system nondeterminism due to workload timing uncertainties such as release time and execution time.

For example, for a system with two cores, any time before A_1^{min}, no core can be available for a job in the ready queue. Similarly, any time before A_2^{min}, only one of the cores can be possibly available. For the time interval $[A_2^{min}, A_2^{max}]$, two of the cores are possibly available. From time A_2^{max}, two of the cores are certainly available.

It is important to note that the system availability interval does not describe the availability of a specific core. Therefore, for interval $A_1 = [A_1^{min}, A_1^{max}]$, the one core that can possibly be available at time A_1^{min} may not be the same as the one core that is certainly available at A_1^{max}.

Example 2. Consider a system with $m = 2$ cores and suppose there are two jobs scheduled on the cores with finish time intervals $[5, 10]$ and $[4, 12]$. In this example, $t = 4$ is the earliest time when one of the m cores is possibly available for the next ready job. Similarly, from time 5, two cores can possibly be available. At $t = 10$, at least one of the m cores is certainly available and at $t = 12$, two cores will be available. Therefore, the system availability interval is calculated as $A_1 = [4, 10]$ and $A_2 = [5, 12]$.

Edges

The set of edges E, where each edge $e = (v, v', J)$ between system states (vertices) v and v' describes a scheduling decision. An edge e describes a scenario where job J is scheduled considering availability at vertex v. The resultant state is described in vertex v' based on the system availability after scheduling J.

Paths

A path P starting from vertex v_1 and ending with vertex v_i is a possible sequence of scheduling decisions from initial system state v_1 to v_i. The set of jobs dispatched on a path P is denoted as $\mathcal{J}_P$.

4.1.1. Schedule abstraction graph construction

The schedule abstraction graph [1], [15], [36] is constructed with a breadth-first approach. Once the initial state v_1 is added, all possible scheduling decisions are considered based on system availability intervals, the set of ready jobs, and the priority of a job amongst the ready jobs. For each of these scheduling decisions, an edge is created connecting v_1 with the vertex representing the state after the scheduling decision. This phase is known as the expansion phase. Then, all of these new states are explored in a breadth-first approach until all jobs are considered in each path. To avoid the state space explosion, a merge phase is performed where newly created states after the expansion phase are assessed to determine the possibility of merging states. Two states are merged if any possible execution scenario that can be explored from the selected two states can still be explored after they are merged. The following sections will discuss the schedule abstraction graph phases in more detail.

Expansion phase

The objective of the expansion phase is to explore the outcome of different scheduling decisions based on a system state. In this phase, (one of) the shortest path(s) P in the SAG (connecting v_1 to v_p) is explored by considering all jobs that can be a direct successor of state v_p. A job J' is said to be a direct successor of a state v_p if there exists an execution scenario in which job J' is dispatched after state v_p and before any other job. A new state is created for each of these jobs by adding an edge to the SAG, connecting the new state v'_p and the terminal vertex of path P, v_p.

To determine the direct successors of state v_p, $\mathcal{R}^P$ a set of potentially ready jobs for system state v_p is identified. $\mathcal{R}^P$ is defined as follows (derived from [1])

$$\mathcal{R}^P \triangleq J_i \in \mathcal{J} \setminus \mathcal{J}^P. \tag{4.1}$$

For each ready job $J_i \in \mathcal{R}^P$, we find the earliest and latest times it can start after state v_p. These times are called the earliest start time (EST) and the latest start time (LST), denoted as $\mathrm{EST}_i(v_p)$ and $\mathrm{LST}_i(v_p)$. A job is eligible as a direct successor of state v_p if its $\mathrm{EST}_i(v_p)$ is no later than its $\mathrm{LST}_i(v_p)$, i.e. if

$$\mathrm{EST}_i(v_p) \leq \mathrm{LST}_i(v_p). \tag{4.2}$$

Definition 2 (From [1])**.** A job $J_i \in \mathcal{R}^P$ is a direct successor of v_p only if inequality (4.2) holds.

As a job cannot start execution before its earliest release time or before at least one of the cores becomes available, the earliest time that J_i can start its execution after state v_p is defined as

$$\mathrm{EST}_i(v_p) = max\{r_i^{min}, A_1^{min}(v_p)\}. \tag{4.3}$$

The latest start time of a job after the system states v_p is decided by two properties of the scheduler: **(i)** The scheduler decides the next job to be dispatched based on the JLFP scheduling policy, and **(ii)** the scheduler follows a work-conserving policy. According to property (i), a job can be a direct successor of v_p if it can start its execution before a higher-priority job is certainly released. the upper bound on the certain release of a higher-priority job for J_i is defined as follows[1]

$$t_{high} \triangleq \min_{\infty}\{r_x^{max} \mid J_x \in \mathcal{R}^P \wedge p_x < p_i\}. \tag{4.4}$$

Based on eq. (4.4), the latest time instant that J_i can start executing after state v_p is $t_{high} - 1$. If J_i is the highest priority job in $\mathcal{R}^P$ (i.e. $p_i = \min p_x \mid J_x \in \mathcal{R}^P$), then t_{high} is set to ∞ and it does not bound the latest start time of J_i.

As the scheduler follows a work-conserving scheduling policy (based on the property (ii)), the second upper bound on the latest start time for job J_i after state v_p is t_{wc} [1], the time at which the core is certainly available and a possibly ready job is certainly released. As the scheduler will dispatch the highest-priority job among the certainly released jobs in $\mathcal{R}^P$, when at least one of the cores is certainly available, the scheduler will assign the ready job. Thus, job $J_i \in \mathcal{R}^P$ will not be dispatched next after v_p at any time later than t_{wc}. Nasri et al. [1] defines t_{wc} as

$$t_{wc} \triangleq \max\{\mathrm{A}_1^{max}(v_p),\ \min\{r_x^{max} \mid J_x \in \mathcal{R}^P\}\}. \tag{4.5}$$

Considering the two upper bounds from eq. (4.4) and eq. (4.5), the latest time instant at which J_i can start such that J_i is a direct successor job of state v_p is

$$\mathrm{LST}_i(v_p) = \min\{t_{wc},\ t_{high} - 1\}. \tag{4.6}$$

If job $J_i \in \mathcal{R}^P$ satisfies inequality (4.2), J_i is dispatched next after v_p and its earliest and latest finish times are [1]

$$\begin{aligned} \mathrm{EFT}_i(v_p) &= \mathrm{EST}_i(v_p) + C_i^{min}, \\ \mathrm{LFT}_i(v_p) &= \mathrm{LST}_i(v_p) + C_i^{max}. \end{aligned} \tag{4.7}$$

Once the job J_i is dispatched after v_p, a new system state v_p' is created to show the system state with the selected scheduling decision. To calculate the system core availability for state v_p', two lists called PA and CA are created. PA and CA contain lower and upper bounds on the time instants at which each number of cores becomes possibly and certainly available at system state v_p, respectively. PA and CA are calculated as follows [1]

$$\begin{aligned} \mathrm{PA} &\triangleq \{\max\{\mathrm{EST}_i(v_p), \mathrm{A}_x^{min}(v_p)\} \| 2 \leq x \leq m\} \cup \mathrm{EFT}_i, \\ \mathrm{CA} &\triangleq \{\max\{\mathrm{EST}_i(v_p), \mathrm{UA}_x^{max}(v_p)\} \| 2 \leq x \leq m\} \cup \mathrm{LFT}_i. \end{aligned} \tag{4.8}$$

After creating the PA and CA list, both lists are sorted in non-decreasing order and assigned to the core availability as follows [1]

$$\forall x, 1 \leq x \leq m, A_x(v_p') \leftarrow [\mathrm{PA}_x, \mathrm{CA}_x]. \tag{4.9}$$

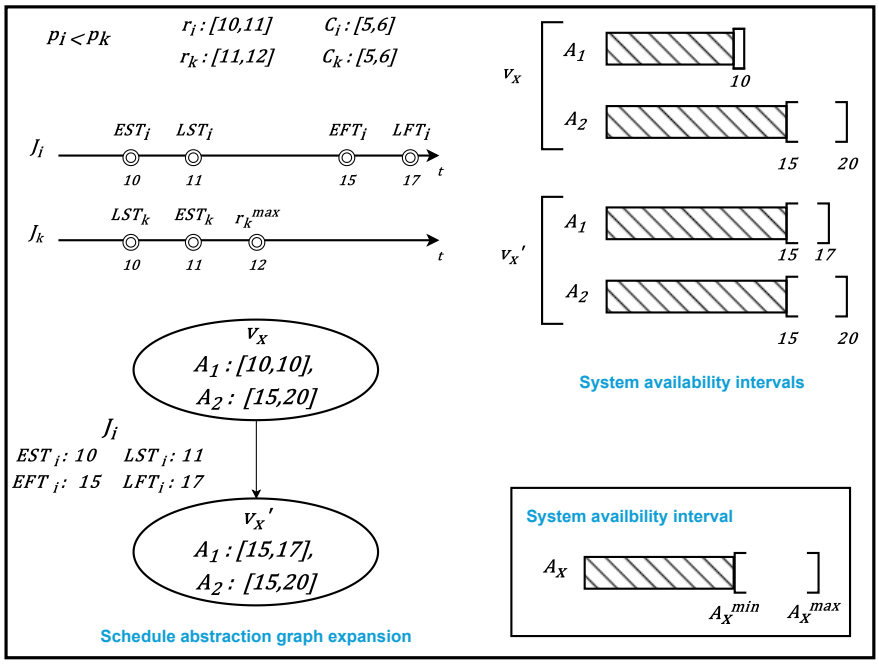


Figure 4.1: Expansion of system state v_x

Example 3. Consider a job set $\mathcal{J}$ with n jobs scheduled on a system with two cores. A system state v_x has set of system availability intervals $A(v_x)$, where

$A_1(v_x) = [10, 10]$ and $A_2(v_x) = [15, 20]$. The set of jobs dispatched on a path connecting the initial state v_1 and v_x is $\mathcal{J}^X$ such that $\mathcal{J}^X = \mathcal{J} \setminus \{J_i, J_k\}$. Figure 4.1 shows the expansion of the system state v_x to state v'_x. Jobs J_i and J_k are potentially ready jobs at state v_x with release time uncertainty. As one core is possibly and certainly available at $t = 10$, based on the earliest release time of each job, the earliest start time $\mathrm{EST}_i(v_x)$ is 10 while $\mathrm{EST}_k(v_x)$ is 11. Since J_i is the highest priority in $\mathcal{R}^P$, t_{high} for J_i is ∞. Thus the latest start time for J_i ($\mathrm{LST}_i(v_x)$) is bounded by $t_{wc} = 11$ (according to eqs. (4.5) and (4.6)). For the job J_k, $t_{high} - 1$ is 10 (according to eq. (4.4)) and $t_{wc} = 11$ (according to eq. (4.5)). Based on eq. (4.6), the latest start time for J_k ($\mathrm{LST}_k(v_x)$) is 10, as at time 11, a higher priority job J_i will certainly claim the system availability $A_1(v_x)$.

The inequality (4.2) holds only for J_i, as $\mathrm{EST}_k(v_x)$ is greater than $\mathrm{LST}_k(v_x)$. Therefore, J_i is the only direct successor of state v_x. This results in system states v'_x connected to v_x with an edge (v_x, v'_x, J_i). The set of system availability intervals for state v'_x is calculated with eqs. (4.7) to (4.9).

Similar to the example 3, each of the shortest paths is further explored in the expansion phase.

Merge phase

A naive state expansion of the schedule abstraction graph can result in a state space explosion for a job set with high timing uncertainty [1]. To reduce the growth speed of the state space exploration, a merge phase is performed after the expansion phase to merge the leaf vertices of two paths whose future is the same or can be explored as a single state. As the expansion phase relies on core availability, all future system states are reachable if sets of dispatched jobs for two states are equal and if all of their core availability intervals overlap. As defined by Nasri et al. [1], the criteria for safe state merge is as follows

Rule 1. *Two states v_p and v_q can be merged if $\mathcal{J}^P = \mathcal{J}^Q$ and $\forall x, 1 \leq x \leq m$, $A_x(v_p) \cap A_x(v_q) \neq \emptyset$*

the set of system-availability interval $A(v_z)$ in the merged state v_z include the availability intervals of both v_p and v_q. The set of system-availability interval $A(v_z)$ is defined as (from [1])

$$A_x(v_z) = [\min A_x^{min}(v_p), A_x^{min}(v_q), \max A_x^{max}(v_p), A_x^{max}(v_q)] \quad \forall x, 1 \leq x \leq m. \tag{4.10}$$

The SAG construction algorithm [1] alternates between the expansion and merge phases until all paths in the graph consider all jobs in the job set, or any paths encounter a deadline miss. The correctness of the schedule abstraction graph can be assessed based on theorem 1 by Nasri et al. [1] which states that for any possible execution scenario, there exists a path in the schedule abstraction graph that represents the schedule of all jobs in the given scenario.

Theorem 1. *For any execution scenario such that a job $J_i \in \mathcal{J}$ finishes at some time t, there exists a path $P =< v_1, \ldots, v_p, v'_p >$ in the schedule-abstraction graph such that J_i is the label of the edge from the state v_p to the state v'_p and $t \in [\mathrm{EFT}_i(v_p), \mathrm{LFT}_i(v_p)]$*

4.2. Limitations of SAG

A schedule abstraction graph considers all different execution scenarios with possible job orderings to determine the best-case and worst-case finish time for a job. While exploring different states, SAG also analyzes the schedulability of the job set by identifying the possible execution scenario where the job's latest finish time is higher than the job's absolute deadline. In energy-aware scheduling, SAG can be used to analyze the schedulability of the job set with assigned job speeds.

The power minimization problem is NP-complete for the dynamic voltage and frequency scaling (DVFS) model with a discrete arbitrary number of speeds [42]. Thus, it is important to minimize the number of jobs considered for speed readjustment in energy-aware scheduling. To effectively identify a set of jobs that affects the response time of a job J, a connection needs to be identified between job J and other previously scheduled jobs while considering the impact of different speeds on each job. The schedule abstraction graph implementation by Nasri et al. [1] considers a fixed-speed exploration. Therefore, SAG does not explore all possible execution scenarios resulting from varying job execution speeds.

This leads to the following sub-questions to the main research question.

SQ 1. How to extend the schedule abstraction graph to effectively explore all possible execution scenarios for a given set of discrete speeds.

SQ 2. For any scheduled job $J \in \mathcal{J}$ in the SAG, how to identify the set of previously scheduled jobs that impact the response time of job J.

SQ 3. How to use the information provided by the SAG to effectively assign an energy-efficient speed to each scheduled job.

To overcome the limitations of the schedule abstraction graph, we extend the SAG to explore all possible execution scenarios with a given discrete speed set and identify connections between the jobs. Finally, we use the connection between jobs to identify energy-aware speeds for each job while ensuring the timing constraints (if possible).

5

Energy aware scheduling

This chapter discusses the proposed solution for energy-aware speed assignment for jobs scheduled using a job-level fixed priority (JLFP) scheduling policy, such as G-NP-EDF. The solution uses a Schedule Abstraction Graph (SAG) [1] to explore all possible scheduling scenarios and verify the schedulability of the job set with energy-aware speeds. Figure 5.1 describes the approach for energy-aware speed assignment. The solution assigns a speed to each job in the job set to reduce overall energy consumption while maintaining the schedulability of the job set.

In the preprocessing stage, speeds that would certainly result in an unschedulable job set are removed by pruning the set of possible speeds for each job to reduce the exploration space for speed assignment. Additionally, an initial infeasibility check is performed based on the available valid speeds (as defined in definition 3) to determine the schedulability of the job set for the provided speed settings. After preprocessing, the SAG is generated to explore the scheduling scenario in which all jobs are executed at their lowest valid speed. This is achieved by adjusting the execution time according to the lowest valid speed (as per eq. (3.2)). If the SAG completes the exploration without any deadline violations, the job set is deemed schedulable at the selected lower speeds, resulting in reduced energy consumption. However, if a deadline violation occurs, an ultimate Schedule Abstraction Graph (ultimate SAG) is generated to explore all execution scenarios considering all valid speeds.

Based on the scheduling scenarios in the ultimate SAG, the set of connected jobs that influence the start and finish times of the job that violated its deadline is identified. After identifying these connected jobs, causal links are generated, representing sets of connected jobs within an execution scenario in the ultimate SAG. An energy-aware speed readjustment is performed for each unique causal link to resolve the deadline violation. If all causal links are explored without achieving a successful speed readjustment, or if the analysis runtime exceeds the timeout threshold, a timeout speed readjustment is performed to assess the schedulability of the job set. If this timeout

speed readjustment is unsuccessful, the job set is deemed unschedulable.

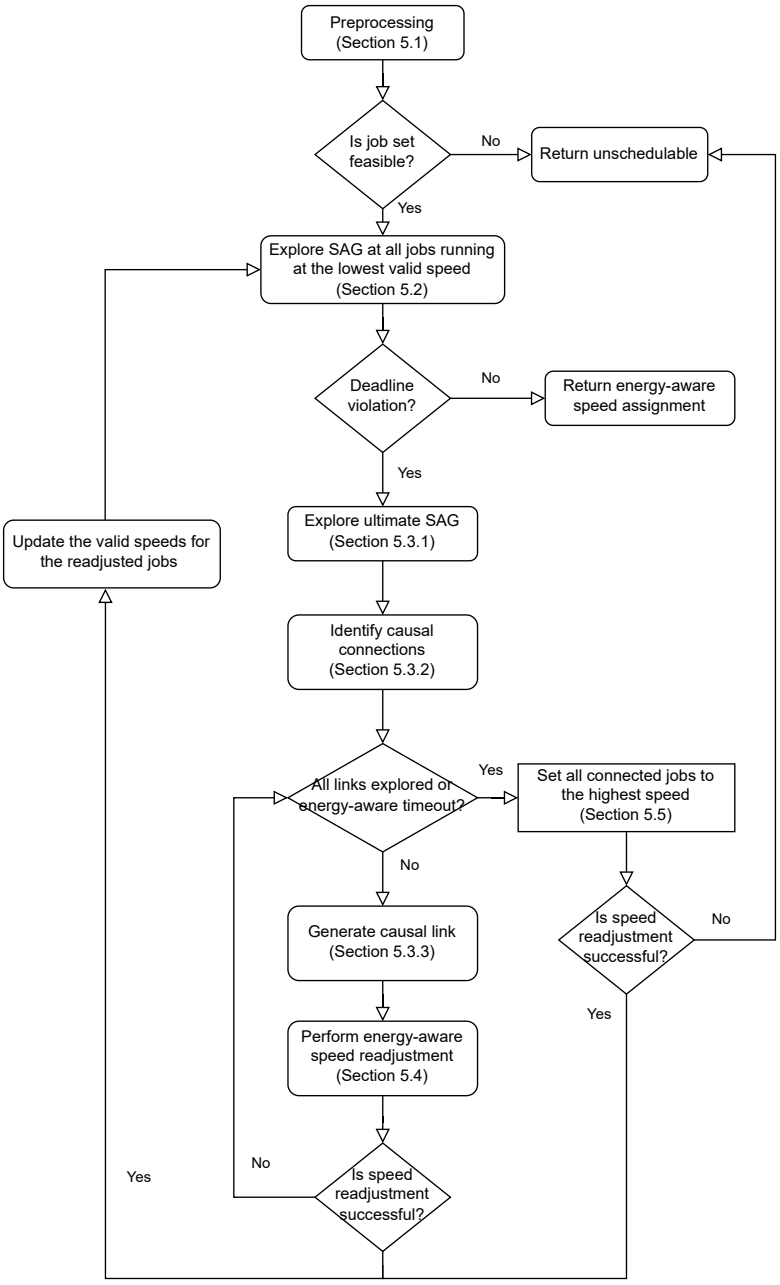


Figure 5.1: Energy aware speed assignment with SAG

Following a successful speed readjustment, valid speeds are recalculated based on the results of speed readjustment, and SAG exploration continues until the exploration is complete. If the job set is schedulable when all

jobs are executed at the highest speed under the selected JLFP scheduling policy, the proposed solution provides an energy-aware speed assignment while maintaining schedulability. The following sections discuss all stages of energy-aware scaling in detail.

5.1. Preprocessing

The goal of energy-aware hard real-time scheduling is to reduce the energy consumption of the system by executing jobs at a lower speed while ensuring the timing requirement of the scheduled job set. As the speed of the core is reduced, the execution time of a job scheduled on the core increases (according to eq. (3.2)). The increased execution time for a job can result in an unschedulable job set.

Example 4. Consider a job J_i with release time (both r_i^{min} and r_i^{max}) at $t = 0$ and a deadline of $t = 10$. Job J_i can be scheduled with speeds 0.5, 0.75, and 1.0 resulting in execution time (both C_i^{min} and C_i^{max}) 12, 8, and 6 units of time, respectively.

The earliest time J_i can start its execution on an available core is its release time, $t = 0$. If a core is available at $t = 0$, job J running at speed 0.75 or 1.0 can complete the execution before its deadline at $t = 10$. However, J_i running at speed 0.5 results in an execution time of 12 units, which exceeds the deadline, leading to a guaranteed deadline violation.

The search space for a feasible energy-aware speed assignment (i.e., an energy-aware speed assignment that does not violate the timing constraints of the jobs) can be reduced by excluding any speed for a job that would certainly result in deadline violations. Therefore, a preprocessing stage performs an initial infeasibility check for a job set based on the system speed configurations. During this stage, a set of potentially valid speeds is identified for each job in the job set. We define a potentially valid speed for a job as follows

Definition 3 (**Potentially valid speed**)**.** For a job $J_i \in \mathcal{J}$, any core speed $\mathcal{S} \in \{\mathcal{S}^{crtl}, \ldots, \mathcal{S}^{max}\}$ that results in $C_i^{max}(S) \leq d_i - r_i^{max}$ is a potentially valid speed.

Lemma 1. *For a job $J_i \in \mathcal{J}$, any core speed $\mathcal{S} \in [\mathcal{S}^{crtl}, \mathcal{S}^{max}]$ that results in $C_i^{max}(S) > d_i - r_i^{max}$ will certainly make the job unschedulable.*

Proof. As $J_i \in \mathcal{J}$ can be released as late as r_i^{max}, there exists an execution scenario where J_i is released at r_i^{max}. Considering a scenario when J_i is released at r_i^{max}, r_i^{max} is the earliest time J_i can start its execution. The execution time of J_i at speed S can be as long as $C_i^{max}(S)$. Therefore, there exists an execution scenario when the execution time of J_i at speed S is $C_i^{max}(S)$. As the finish time of a job depends on the start time and the execution time of the job, when the start time of J_i is r_i^{max} and the finish time of J_i resulting from $C_i^{max}(S)$ is later than d_i, then there will be an execution scenario where J_i violates the deadline. □

After purging away all trivially non-valid speeds for a job, we create a speed

space for the job, as defined below

Definition 4 (**speed space**)**.** For a job $J_i \in \mathcal{J}$, speed space (Z_i) is the set of potentially valid speeds for the job J_i, i.e.,

$$Z_i = \left\{ S \mid S \in \{\mathcal{S}^{crtl}, \ldots, \mathcal{S}^{max}\} \wedge C_i^{max}(S) \leq d_i - r_i^{max} \right\}.$$

For energy-aware speed assignment for a job, speeds are only selected from its speed space.

Based on the speed space for each job, the initial schedulability for the job set can be assessed. As proven by lemma 2, if there is any job in the job set with an empty speed space, no speed assignment can result in a feasible schedule.

Lemma 2. *If the speed space for a job $J_i \in \mathcal{J}$ is empty, then $\mathcal{J}$ is certainly unschedulable.*

Proof. According to definition 4, speed space is the set of potentially valid speeds for the job J_i. When the speed space is empty, there are no potentially valid speeds for J_i. Therefore, there exists no speed that can result in the execution time of the job such that the job finishes before the deadline (based on lemma 1). As the job set is deemed schedulable only if no jobs result in a deadline miss if all speeds are invalid for a job $J_i \in \mathcal{J}$, $\mathcal{J}$ is certainly unschedulable. □

Once a speed space is generated for each job in the job set with at least one speed in the speed space, the preprocessing stage is complete with speed assignment search space for each job limited to valid speeds.

5.2. Creating schedule abstraction graph at the lowest speeds

After the preprocessing is completed, all job execution times are calculated based on the lowest speed in the job's speed space, and the schedule abstraction graph is created. As a lower speed in the speed space of a job results in lower energy consumption, all jobs running at the lowest speed in their speed space generate an energy-aware schedule with reduced energy consumption.

Algorithm 1 presents the algorithm for SAG generation with the lowest speeds. In line 1, a set of variables is initialized to track the earliest and latest finish times (denoted by FT_i^{min} and FT_i^{max}, respectively) for each job across all execution scenarios explored so far. These variables are updated whenever job J_i is dispatched to a core (lines 12 and 13). We then calculate The execution time intervals for each job according to the lowest speed in their speed space to reflect the times resulting from the execution at the lowest speed (line 2).

As discussed in chapter 4, a schedule abstraction graph is generated in an iterative breadth-first approach until all paths represent a valid schedule for

Algorithm 1 SAG exploration at energy-aware speeds

Input: Job set $\mathcal{J}$
Output: Schedulability result for the job set $\mathcal{J}$

1: $\forall J_i \in \mathcal{J}, \mathrm{FT}_i^{min} \leftarrow \infty, \mathrm{FT}_i^{max} \leftarrow 0$;
2: $\forall J_i \in \mathcal{J}, C_i^{min} \leftarrow C_i^{min}(\min\{S \mid S \in Z_i\})$, $C_i^{max} \leftarrow C_i^{max}(\min\{S \mid S \in Z_i\})$;
3: Initialize G by adding $v_1 = (\{[0,0], \ldots, [0,0]\})$;
4: **while** $\exists$ path P from v_1 to a leaf vertex s.t. $|P| < |\mathcal{J}|$ **do**
5: $P \leftarrow$ the shortest path from v_1 to a leaf vertex v_p;
6: $\mathcal{R}^P \leftarrow$ set of ready jobs obtained by eq. (4.1);
7: **for** each job $J_i \in \mathcal{R}^P$ **do**
8: **if** J_i can be dispatched after v_p according to eq. (4.2) **then**
9: Calculate PA and CA lists with eq. (4.8);
10: Calculate v_p core availability with eq. (4.9);
11: Build v'_p;
12: $\mathrm{FT}_i^{min} \leftarrow \min\{\mathrm{EFT}_i(v'_p), \mathrm{FT}_i^{min}\}$;
13: $\mathrm{FT}_i^{max} \leftarrow \max\{\mathrm{LFT}_i(v'_p), \mathrm{FT}_i^{max}\}$;
14: Connect v_p to v'_p by an edge with label J_i;
15: **while** $\exists$ path Q that ends to v_q s.t. Rule 1 is satisfied for v'_p and v_q **do**
16: Merge v'_p and v_q by updating v'_p using eq. (4.10);
17: Redirect all incoming edges of v_q to v'_p;
18: Remove v_q from V;
19: **end while**
20: **if** $\mathrm{FT}_i^{max} > d_i$ **then**
21: Return unschedulable;
22: **end if**
23: **end if**
24: **end for**
25: **end while**
26: Return schedulable;

all jobs in the job set $\mathcal{J}$. During the expansion phase (lines 5-14), one of the shortest paths P is expanded by generating a new state v_p for each job J_i and adding it to the graph via a directed edge from v_p to v'_p (as explained in section 4.1.1). To prevent search space explosion, a merge phase (as explained in section 4.1.1) is performed, wherein paths with intersecting availability intervals and the same set of jobs are merged (lines 15-19). After dispatching each job, the algorithm checks for any deadline violations (lines 20-22). If the dispatched job meets the deadline constraints, the expansion and merging stages are repeated until all execution scenarios are explored without any deadline violations. If a deadline violation is detected, the job causing the deadline miss is returned, and the result is marked as unschedulable.

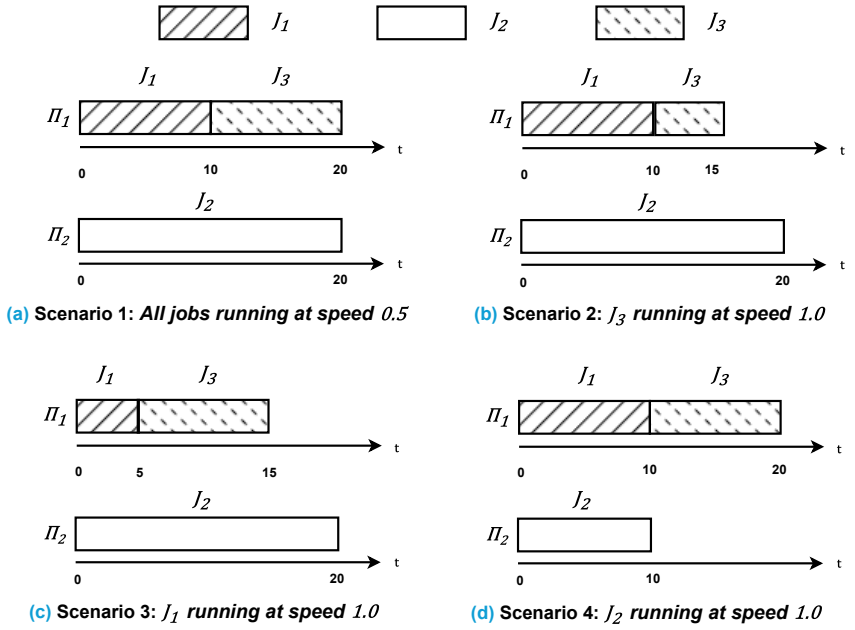
As the finish time for a job J depends on its execution time and start time (according to eq. (4.7)), resolving the deadline miss for J requires adjusting the speed for the job J and the jobs scheduled before J, such that their

combined execution time result in a finish time for J that respects its deadline. To determine which set of jobs can resolve the deadline violation, the connection between these jobs must be established.

5.3. Connection with dispatched job

This section describes an approach for identifying a set of jobs that can resolve a deadline miss.

As discussed in section 5.2, the deadline violation can be resolved by readjusting the speed of scheduled jobs. Since the platform has multiple cores, jobs can be executed concurrently. Therefore, changing a job's execution time does not necessarily affect the response time for all subsequently scheduled jobs.



(a) Scenario 1: *All jobs running at speed 0.5*

(b) Scenario 2: J_3 *running at speed 1.0*

(c) Scenario 3: J_1 *running at speed 1.0*

(d) Scenario 4: J_2 *running at speed 1.0*

Figure 5.2: Impact of speed scaling on a multi-core platform

Example 5. Consider three jobs scheduled on a system with two cores under the JLFP scheduling policy. Figure 5.2 shows the resultant scheduling scenarios. Job J_1 and J_3 are scheduled on core π_1 while job J_2 is scheduled on core π_2. Suppose the speed space for all three jobs is $\{0.5, 1.0\}$ and jobs J_1 and J_3 have the same execution times 10 and 5 units, resulting from executing the jobs at 0.5 and 1.0 speed, respectively. The execution times for job J_2 are 20 and 10 units at speeds 0.5 and 1.0, respectively. All jobs are released at time $t = 0$ with the priority order $p_1 < p_2 < p_3$ (lower value indicates higher priority). Figure 5.2(a) shows the execution scenario when all jobs are executed at the speed 0.5. In this example, executing job J_3 at speed 1.0 reduces its finish time (shown in fig. 5.2(b)) while executing job J_1 at speed 1.0 reduces the start time for job J_3, resulting in an earlier finish

time for J_3 (shown in fig. 5.2(c)). On the other hand, running job J_2 at the highest speed does not impact the finish time for job J_3 as the time J_3 can start its execution is not affected (shown in fig. 5.2(d)).

As computation complexity for identifying the combination of speed to resolve the deadline miss depends on the number of previously scheduled jobs, it is important to limit the search space to the previously scheduled jobs that affect the response time for the deadline miss job. Therefore, the connection between the jobs needs to be established to optimize the speed scaling.

In the schedule abstraction graph, when a new vertex v'_p is generated by expanding the path connecting the initial vertex v_1 and leaf vertex v_p, the system availability interval for vertex v'_p is calculated based on the finish time of the job scheduled till the vertex v_p. Similarly, the job's start time in state v'_p depends on the system availability interval of v'_p (according to eqs. (4.3) and (4.6)). Therefore the start time interval of a job J (dispatched at state v'_p) is due to the system availability provided by previously dispatched job(s) whose finish time influences the system availability interval $A_1(v'_p)$ (system interval when one core is available). Hence, a change in the finish time of job(s) that influences $A_1(v'_p)$ can change the finish time for J.

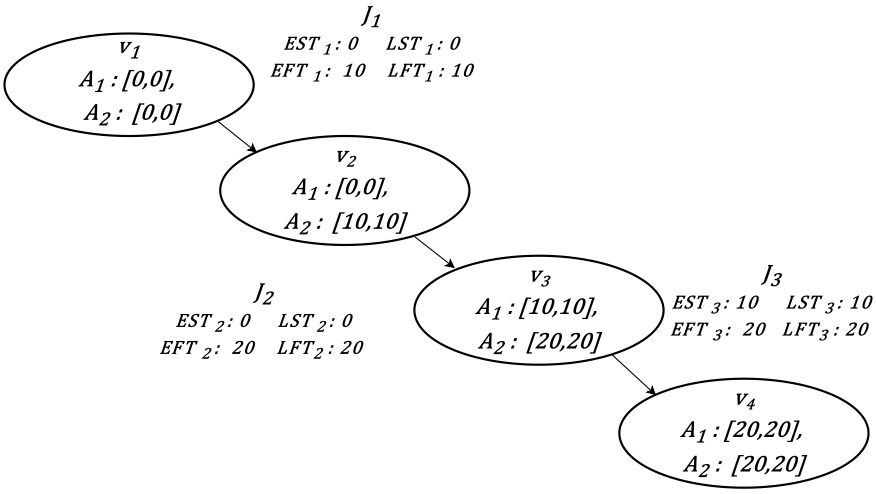


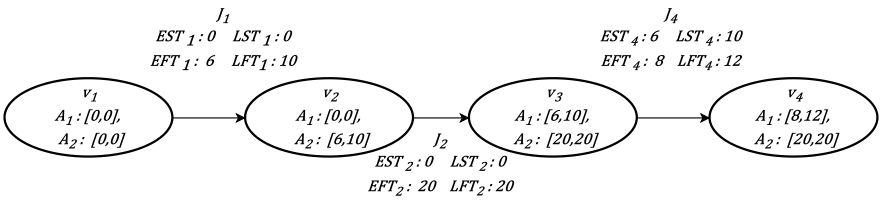
Figure 5.3: Schedule abstraction graph resulting from all jobs in example 5 running at speed 0.5

Example 6. Consider the system and workload from example 5. Figure 5.3 shows the resulting schedule abstraction graph with system availability intervals when all jobs are running at the lowest speed (0.5). As observed in fig. 5.3, the start time of job J_3 overlaps with the finish time of job J_1 as the system availability interval $A_1(v_3)$ depends on the finish time of J_1 while $A_2(v_3)$ depends on the finish time of J_2. If J_1 is executed at speed 1.0, the finish time of J_1 is reduced. This provides an earlier start time for J_3 which reduces the finish time for J_3 (as observed in fig. 5.2(b)).

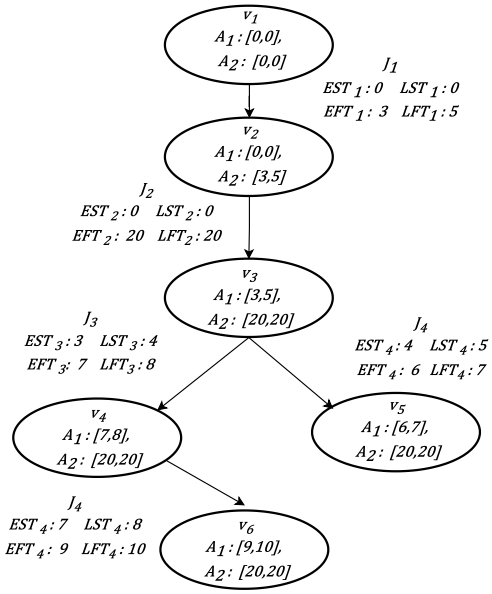
One of the limitations of the schedule abstraction graph is that it can explore the execution scenario based on each job set at a fixed speed. Although the connection between jobs can be identified based on analyzing the start and finish time with all explored scenarios, changing the speed of any job can change the execution scenarios or even create new scenarios.

(a) Job set with execution time intervals at speed 0.5 and 1.0

Job	Release time	Execution time at speed 1.0		Execution time at speed 0.5		Deadline	Priority
		min	max	min	max		
J_1	0	3	5	6	10	10	High
J_2	0	10	10	20	20	20	High
J_3	0	2	2	4	4	10	Low
J_4	4	1	1	2	2	9	High



(b) Scenario 1: *All jobs running at speed 0.5*



(c) Scenario 2: J_1 *running at speed 1.0*

Figure 5.4: Impact of speed change on execution scenarios

Example 7. Consider a job set with four jobs scheduled on a system with two cores. The fig. 5.4(b) shows the schedule abstraction graph expansion when all jobs are executed at the lowest speed. Jobs J_1 and J_2 are the highest priority ready jobs at time 0 that are executed on two available cores. The earliest time when at least one core is possibly available is at time $t = 6$, and it will certainly be available after $t = 10$. Considering core availability $A_1(v_3)$, the next scheduled job certainly will be J_4 (highest priority released job). This job violates its deadline constraints if the core becomes certainly available any time after $t = 7$.

If J_1 is executed at the speed 1.0 (to resolve the deadline miss), a new execution scenario is generated where $A_1^{min}(v_3)$ is at $t = 3$ and J_3 can start its execution on this core as J_4 is released at time 4. As observed in fig. 5.4(c), the newly generated execution scenario adds a new job ordering where J_3 can be scheduled before J_4.

Hence, changing the speed can lead to iterative sub-problems of speed readjustment for resolving deadline misses, as each adjustment can introduce new execution scenarios and job ordering. To effectively find the connection between jobs, considering all speeds and resulting execution scenarios, the schedule abstraction graph needs to be updated to consider the impact of different speeds on state space exploration. To solve this problem, we present an ultimate schedule abstraction graph.

5.3.1. Ultimate schedule abstraction graph

The proposed ultimate schedule abstraction graph (ultimate SAG) extends the concept of the schedule abstraction graph by exploring all possible execution scenarios resulting from all speeds in the speed space for each job in the job set. A schedule abstraction graph explores new states from a system state v_p based on the system availability intervals after v_p. The set of system availability intervals after the state v_p is calculated according to the execution times of the job J_i, scheduled in the state v_p. To explore states considering all possible speeds for J_i, we set the execution time interval for J_i to the the ultimate execution interval, UET_i, which is calculated as follows

$$\begin{aligned} \mathrm{UET}_i^{min} &= C_i^{min}(\max\{S \mid S \in Z_i\}), \\ \mathrm{UET}_i^{max} &= C_i^{max}(\min\{S \mid S \in Z_i\}). \end{aligned} \tag{5.1}$$

To identify all possible execution scenarios till the deadline miss job J_d is scheduled, ultimate SAG is generated until all paths connecting the initial state v_1 till all leaf vertices contain the job J_d. The algorithm for ultimate SAG generation is explained in algorithm 2.

To generate an ultimate schedule abstraction graph, the execution time interval for all jobs in the job set is set to the ultimate execution time (according to the eq. (5.1)). As discussed in the example 6, job start and finish times can provide insight into the connection between the jobs. Therefore, the ultimate start time UST_i and ultimate finish time UFT_i intervals are calculated for all jobs explored in the ultimate SAG (lines 12-15). The expansion (lines 7-16) and merge phase (lines 17-21) in the ultimate SAG is the same as the

Algorithm 2 Ultimate SAG construction algorithm

Input: Job set $\mathcal{J}$ with the deadline miss job $J_d \in \mathcal{J}$
Output: Schedule graph $G = (V, E)$

$\forall J_i \in \mathcal{J}, \mathrm{UFT}_i^{min} \leftarrow \infty, \mathrm{UFT}_i^{max} \leftarrow 0, \mathrm{UST}_i^{min} \leftarrow \infty, \mathrm{UST}_i^{max} \leftarrow 0$;
$\forall J_i \in \mathcal{J}, C_i \leftarrow \mathrm{UET}_i$;
Initialize G by adding $v_1 = (\{[0,0], \ldots, [0,0]\})$;
while $\exists$ path P from v_1 to a leaf vertex s.t. $J_d \notin \mathcal{J}^P$ **do**
    $P \leftarrow$ the shortest path from v_1 to a leaf vertex v_p;
    $\mathcal{R}^P \leftarrow$ set of ready jobs obtained by eq. (4.1);
    for each job $J_i \in \mathcal{R}^P$ **do**
        if J_i can be dispatched after v_p according to eq. (4.2) **then**
            Calculate PA and CA lists with eq. (4.8);
            Calculate v_p core availability with eq. (4.9);
            Build v'_p;
            $\mathrm{UFT}_i^{min} \leftarrow \min\{\mathrm{EFT}_i(v'_p), \mathrm{UFT}_i^{min}\}$;
            $\mathrm{UST}_i^{min} \leftarrow \min\{\mathrm{EST}_i(v'_p), \mathrm{UST}_i^{min}\}$;
            $\mathrm{UFT}_i^{max} \leftarrow \max\{\mathrm{LFT}_i(v'_p), \mathrm{UFT}_i^{max}\}$;
            $\mathrm{UST}_i^{max} \leftarrow \max\{\mathrm{LST}_i(v'_p), \mathrm{UST}_i^{max}\}$;
            Connect v_p to v'_p by an edge with label J_i;
            while $\exists$ path Q that ends to v_q s.th. Rule 1 is satisfied for v'_p and v_q **do**
                Merge v'_p and v_q by updating v'_p using eq. (4.10);
                Redirect all incoming edges of v_q to v'_p;
                Remove v_q from V;
            end while
        end if
    end for
end while

expansion and merge phase in the SAG. The ultimate SAG is explored until all leaf vertices contain the job that violated the deadline miss.

The lemma 3 proves that ultimate SAG exploration explores all possible execution scenarios, considering the speed space of the job.

Lemma 3. *Consider a path Q from a vertex v_1 to a leaf vertex v_q in the schedule abstraction graph (SAG) for the job set, $\mathcal{J}$, where each job $J_i \in \mathcal{J}$ is assigned to a single arbitrary speed $S \in Z_i$. Then, any execution scenario that can be observed from Q is also observed in the ultimate schedule abstraction graph for the job set, $\mathcal{J}$.*

Proof. Consider a path Q from v_1 to a leaf vertex v_q in the SAG for the job set, $\mathcal{J}$ such that each job $J_i \in \mathcal{J}_Q$ is assigned to a single arbitrary speed $S \in Z_i$.

According to the eq. (5.1), the bounds on ultimate execution time intervals are calculated based on execution times with the lowest and highest speed in the speed space. Therefore, the execution time of $J_i \in \mathcal{J}_Q$ with speed $S \in Z_i$, $[C_i^{min}(S), C_i^{max}(S)]$ is a subset of its ultimate execution time

$[\mathrm{UET}_i^{min}, \mathrm{UET}_i^{max}]$, i.e, $C_i^{min}(S) \geq \mathrm{UET}_i^{min}$ and $C_i^{max}(S) \leq \mathrm{UET}_i^{max}$.

According to theorem 1, for any possible execution scenario, there exists a path in the SAG (and ultimate SAG) that represents the schedule of all jobs in the given scenario. Therefore, there also exists a path P in the ultimate SAG for an execution scenario where the execution time of jobs is the same as $\forall J_i \in \mathcal{J}_Q$ where$[C_i^{min}(S), C_i^{max}(S)]$. □

5.3.2. Causal connection

This section discusses an approach for identifying the connection between the jobs based on the scenarios explored in the ultimate SAG. We define the connection from a job J to another job that can impact the finish time of J as a causal connection.

Definition 5 (**Causal connection**)**.** Two jobs are considered to be causally connected if considering all speed combinations for the jobs in $\mathcal{J}$, there exists an execution scenario, where these jobs are executed on the same core without any idle time in between. The causal connection is denoted with $\Rightarrow$.

A Causal order is the order of execution for causally connected jobs.

Definition 6 (**Causal order**)**.** Causal order is the temporal order in which causally connected jobs are executed. For causally connected jobs J_1, J_2 $\in \mathcal{J}$, $J_2 \Rightarrow J_1$ indicate that there exists an execution scenario where J_2 is executed after J_1 such that the jobs are causally connected.

As discussed in the example 6, in an execution scenario, an overlap[1] between the start time of a job and the finish time of another job is an indicator for a causal connection. When the start and finish time intervals are calculated considering all possible execution scenarios, the overlap does not necessarily consider the execution order for the jobs.

Example 8. Figure 5.5 shows the ultimate SAG graph for four jobs scheduled on two cores. For ease of explanation, we do not merge the states in this example. The ultimate execution time interval resulting from the job's speed space is also described in fig. 5.5. Job J_1 has multiple dispatch orders in different execution scenarios resulting from release time uncertainty with job J_1. Due to multiple orders, the ultimate start time for J_1 is inflated. Considering overlap condition in example 6 to identify causally connected jobs from J_1, ultimate start time of J_1, UST_1 ($[1, 5]$) overlaps with ultimate finish time of J_3, UFT_3 ($[3, 10]$). Although the inflated ultimate start time of J_1 overlaps with the ultimate finish time of J_3, in every execution scenario J_3 is certainly scheduled after J_1. Therefore, J_3 cannot be causally connected with J_1 as it does not follow the causal order.

To identify the causal connection between two arbitrary jobs in the job set,J_1 and J_2 where $J_2 \Rightarrow J_1$, the order of execution for J_1 and J_2 needs to be established. The ordering of the jobs depends on the number of possible

[1] In the discrete-time model, overlap in the time intervals is only considered for the overlapping interval with more than a single time instance, as at a single time instance, the scheduling decision is fixed.

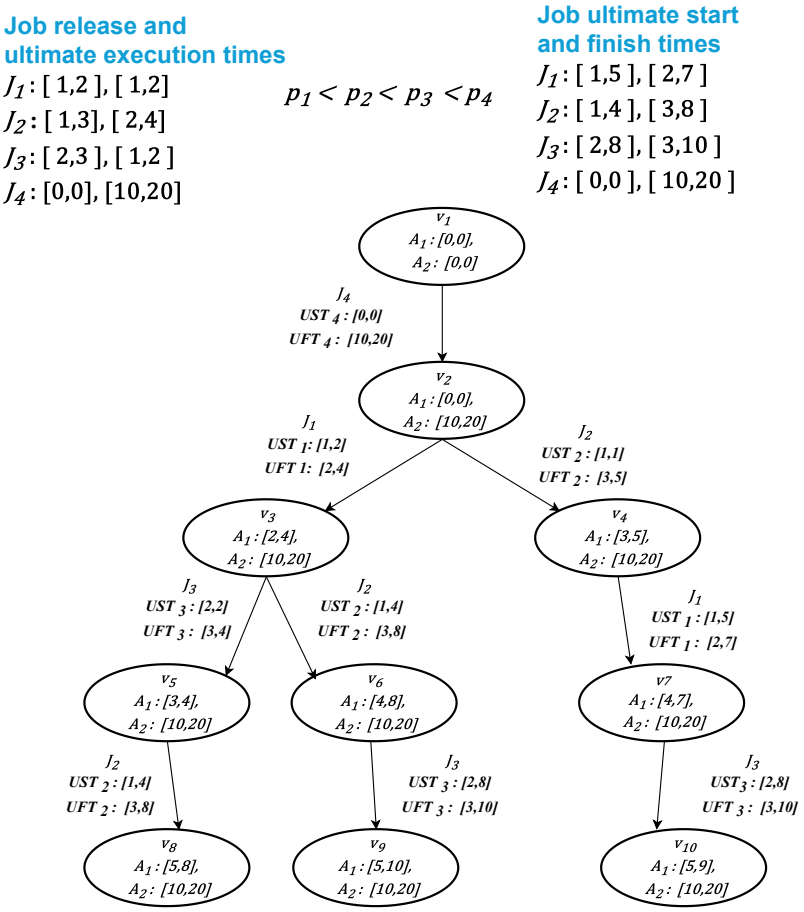


Figure 5.5: Incorrect causal connection between J_1 ad J_3 based on the overlap

direct successors of the state. As the direct successors of a state are identified based on inequality 4.2 and eqs. (4.3) and (4.6), the possibility for a job order can be identified based on the release time interval of the jobs, the priority of the jobs, and the set of system availability intervals for the state. As seen in example 8, ultimate start and finish time overlap does not guarantee the causal order due to inflated ultimate start and finish time interval.

As the JLFP scheduler selects the highest priority ready job, the priority of the jobs can be an indicator of the possible ordering. As observed in example 8, $[UST_3^{min}, UST_3^{max}]$ overlaps with the ultimate finish time interval $[UFT_1^{min}, UFT_1^{max}]$ of a higher priority job J_1. As J_1 has a higher priority than J_3 and certainly releases before r_1^{max}, J_1 executes before J_3 in every execution scenario.

Although in example 8, J_2 has a lower priority than J_1, as seen in ultimate SAG states, v_2, v_4, and v_7, there is an execution scenario where J_2 executes before J_1. At the state v_2, J_2 is considered as a direct successor of v_2 only when J_2 starts its execution as late as when J_1 is not certainly released,

assuming that J_1 is not yet released.

The earliest time J_3 can start its execution, a higher priority job J_1 is certainly released hence J_3 cannot start its execution until J_1 is dispatched.

As observed in example 8, the priority of jobs is an indicator for determining the order of execution of the jobs. For jobs $J_1, J_2 \in \mathcal{J}$,lemmas 4 and 5 shows the conditions to verify if job J_1 can be dispatched before J_2 when UST_2 of job J_2 overlaps with UFT_1 of job J_1. Additionally, lemma 6 shows that even if the UST_2 of job J_2 overlaps with UFT_1 of job J_1 but the conditions from lemmas 4 and 5 are not satisfied, then J_1 is certainly dispatched after J_2 in every execution scenario and cannot provide an earlier start time for J_2.

Lemma 4. *Let $\mathcal{J}^D$ denote a set of jobs that are dispatched in ultimate SAG before the deadline miss job J_d is dispatched in all paths. For jobs $J_1, J_2 \in \mathcal{J}^D$, if $\mathrm{UST}_2 \cap \mathrm{UFT}_1 \neq \emptyset \wedge p_1 < p_2$, then there exists at least one execution scenario in the ultimate SAG where J_1 is scheduled before J_2.*

Proof. Consider two jobs $J_1, J_2 \in \mathcal{J}$ such that $p_1 < p_2$ and $\mathrm{UST}_2 \cap \mathrm{UFT}_1 \neq \emptyset$. Assume that t is a time instance such that $t \in \mathrm{UST}_2 \cap \mathrm{UFT}_1$. As t lies in the overlapping region of UST_2 and UFT_1, there exists a scenario in ultimate SAG where system availability interval allows J_2 to start the execution at time t.

Similarly, an execution scenario exists in ultimate SAG, such that J_1 finishes its execution at time t. In the execution scenario when J_1 finishes its execution at time t, according to eq. (4.7), the start time of J_1 must be before t. As the earliest start time of a job in an execution scenario is calculated by eq. (4.3), the earliest release time of J_1, r_1^{min} must be before t, i.e. $r_1^{min} < t$.

The ultimate SAG and SAG explore all execution scenarios based on the JLFP scheduling policy where the highest priority ready job is selected to execute on an available core. If J_2 starts its execution at t, when a higher priority job, J_1 is potentially released, J_1 must already dispatched when J_2 starts its execution at t or J_1 will start its execution at time t_1 such that $t \leq t_1$ and J_2 start the execution at t_2, where $t_2 \geq t_1$.

Therefore, when $\mathrm{UST}_2 \cap \mathrm{UFT}_1 \neq \emptyset \wedge p_1 < p_2$, then there exists at least one execution scenario in the ultimate SAG where J_1 is scheduled before J_2. □

Lemma 5. *Let $\mathcal{J}^D$ denote a set of jobs that are dispatched in ultimate SAG before the deadline miss job J_d is dispatched in all paths. For jobs $J_1, J_2 \in \mathcal{J}^D$, if $\mathrm{UST}_2 \cap \mathrm{UFT}_1 \neq \emptyset \wedge p_2 < p_1 \wedge \mathrm{UST}_1^{min} < r_2^{max}$, then there exists at least one execution scenario in the ultimate SAG where J_1 is scheduled before J_2.*

Proof. Consider two jobs $J_1, J_2 \in \mathcal{J}$ such that $p_1 > p_2$ and $\mathrm{UST}_2 \cap \mathrm{UFT}_1 \neq \emptyset$ with $\mathrm{UST}_1^{min} < r_2^{max}$. According to the JLFP scheduling policy and eq. (4.6), the inequality 4.2 only holds for a lower priority job J_1 when UST_1^{min} is before J_2's certain release time r_2^{max}. As $\mathrm{UST}_1^{min} < r_2^{max}$, there must be at least

an execution scenario where J_1 starts the execution before J_2 is certainly released. □

Lemma 6. *Let $\mathcal{J}^D$ denote a set of jobs that are dispatched in ultimate SAG before the deadline miss job J_d is dispatched in all paths. For jobs $J_1, J_2 \in \mathcal{J}^D$, if $\mathrm{UST}_2 \cap \mathrm{UFT}_1 \neq \emptyset \wedge p_2 < p_1 \wedge \mathrm{UST}_1^{min} \geq r_2^{max}$, then there is no execution scenario in the ultimate SAG where J_1 is scheduled before J_2.*

Proof. Consider two jobs $J_1, J_2 \in \mathcal{J}$ such that $p_1 > p_2$ and $\mathrm{UST}_2 \cap \mathrm{UFT}_1 \neq \emptyset$ with $\mathrm{UST}_1^{min} < r_2^{max}$. According to the JLFP scheduling policy and eq. (4.6), the inequality 4.2 only holds for a lower priority job J_1 when UST_1^{min} is before J_2's certain release time r_2^{max}. As $\mathrm{UST}_1^{min} \geq r_2^{max}$, higher priority job J_2 is certainly released before the earliest time J_1 starts its execution in ultimate SAG. Therefore, there will be no execution scenario in ultimate SAG where J_1 executes before J_2. □

Based on the lemma 6, causal connection between two jobs can be identified based on the Rule 2.

Rule 2 (**Causal connection**). *A job $J_2 \in \mathcal{J}$ is causally connected with a job $J_1 \in \mathcal{J}$ (i.e. $J_2 \Rightarrow J_1$) if $\mathrm{UST}_2 \cap \mathrm{UFT}_1 \neq \emptyset$ and at least one of the following conditions is satisfied:*

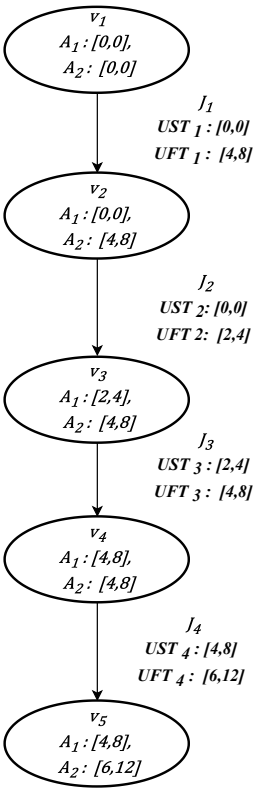
- $p_1 < p_2$ *(based on lemma 4),*
- $\mathrm{UST}_1^{min} < r_2^{max}$ *(based on lemma 5).*

Based on the causal connection between jobs, the set of jobs can be identified that influences the job that missed its deadline.

5.3.3. Causal link

After the causal connections between the jobs are identified considering all execution scenarios in the ultimate SAG, the search space for speed readjustment can be limited to the the sequence of jobs causally linked to the deadline-missed job. In this sequence, each job influences the start time of the subsequent job, thereby reducing the search space to those jobs that directly affect the timing of the missed deadline.

Example 9. Consider four jobs scheduled on two cores. When all jobs are running at the lowest speed in their respective speed space, job J_4 violates its deadline constraint. Figure 5.6 shows the resulting ultimate SAG until J_4 is dispatched in all paths. According to rule 2, causal connections for all jobs are identified, where $J_4 \Rightarrow J_3$, $J_4 \Rightarrow J_1$, and $J_3 \Rightarrow J_2$ while J_1 and J_2 have no causal connection. Based on the causal connections, the finish time of J_4 can be reduced by adjusting the speed of J_4, J_3, and J_1 as the finish time of J_3 and J_1 influences the start time of J_4. Similarly, the finish time of J_3 can be reduced by adjusting the speed of J_2, further improving the finish time of J_4. Therefore, to resolve the deadline violation of J_4, sequences of causally connected jobs $J_4 \Rightarrow J_3 \Rightarrow J_2$ and $J_4 \Rightarrow J_1$ can be used for speed readjustment.



Job	Release time	Ultimate execution time		Deadline	Priority
		min	max		
J_1	0	4	8	10	0
J_2	0	2	4	10	1
J_3	0	2	4	10	2
J_4	0	2	4	10	3

Causal connection

$J_4 \Rightarrow J_3$

$J_4 \Rightarrow J_1$

$J_3 \Rightarrow J_2$

Figure 5.6: Ultimate schedule abstraction graph for workload in example 9

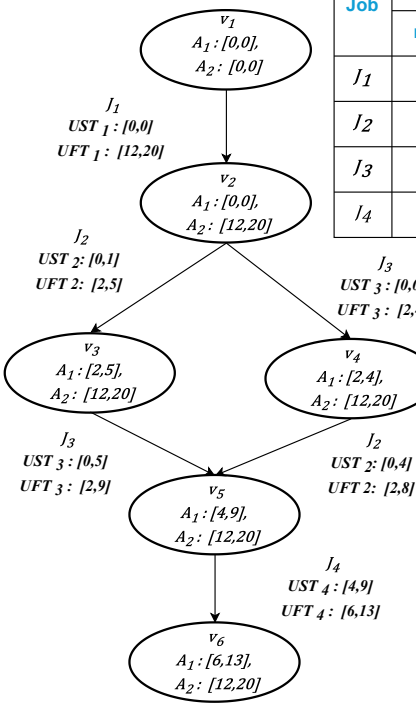
The sequence of causally connected jobs starting from the deadline miss job is called a causal link. A causal link is defined as follows.

Definition 7 (**Causal link**). A causal link is a sequence of jobs executed consecutively on the same core, without any idle time between them, in an execution scenario considering all possible speed combinations, beginning with the job that missed its deadline.

When two jobs are causally connected to each other (resulting from multiple execution orders), the connection needs to be recorded in the causal link only once as another causal link can be generated to indicate an alternate execution order based on another causal connection.

Example 10. Consider four jobs scheduled on two cores. When all jobs are running at the lowest speed in their respective speed space, job J_4 violates its deadline constraint. Figure 5.7 shows the resulting ultimate SAG until J_4 is dispatched in all paths. As observed, J_2 and J_3 have multiple execution orders due to timing uncertainty at v_2. The causal connections for all jobs are identified according to rule 2, where J_1 has no causal connection. When a causal link is generated based on a sequence of causally connected jobs, $J_4 \Rightarrow J_3 \Rightarrow J_2$, the causal link cannot be extended further even when $J_2 \Rightarrow J_3$ as J_3 is already part of the causal link. Alternatively, considering $J_4 \Rightarrow J_2$,

another causal link $J_4 \Rightarrow J_2 \Rightarrow J_3$ can be generated for the execution order where J_2 is executed after J_3.



Job	Release time		Ultimate execution time		Deadline	Priority
	min	max	min	max		
J_1	0	0	12	20	20	0
J_2	0	1	2	4	10	1
J_3	0	1	2	4	10	2
J_4	2	2	2	4	10	3

Causal connection

$J_4 \Rightarrow J_3$

$J_4 \Rightarrow J_2$

$J_3 \Rightarrow J_2$

$J_2 \Rightarrow J_3$

Figure 5.7: Ultimate schedule abstraction graph for workload in example 10

The objective of the causal link is to identify causally connected jobs to reduce the finish time of the deadline miss job. When a job, J_i is certainly scheduled after the deadline miss job, adding J_i to the causal link cannot resolve the deadline violation.

Example 11. Consider four jobs scheduled on two cores. When all jobs are running at the lowest speed in their respective speed space, job J_3 violates its deadline constraint. Figure 5.8 shows the resulting ultimate SAG until J_3 is present in all paths. Based on the causal connections identified according to rule 2, the finish time of J_3 can be improved by adjusting the speed of J_2. Additionally, the finish time of J_2 can be improved by adjusting the speed of J_4. Considering the causal link $J_3 \Rightarrow J_2 \Rightarrow J_4$, the last job J_4 cannot improve time for J_3 as J_4 has a causal connection to J_3 and J_4 executes after J_3 in every execution scenario. Therefore, adding J_4 to causal link $J_3 \Rightarrow J_2$ does not help resolve the deadline violation.

Lemma 7. *Consider a causal link from deadline miss job J_d till a job J_1 with a causal connection to another job, J_2. If J_2 has a causal connection to J_d but J_d does not have a causal connection to J_2, no execution scenario in ultimate SAG exists where J_2 is dispatched before J_d.*

Job	Release time		Ultimate execution time		Deadline	Priority
	min	max	min	max		
J_1	0	0	12	20	20	0
J_2	0	4	2	4	10	2
J_3	0	1	2	4	6	1
J_4	1	3	2	4	10	3

Causal connection

$J_2 \Rightarrow J_4$

$J_2 \Rightarrow J_3$

$J_3 \Rightarrow J_2$

$J_4 \Rightarrow J_3$

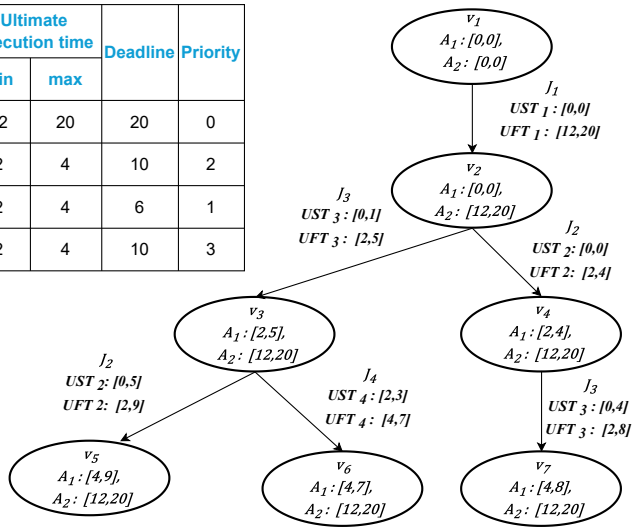


Figure 5.8: Ultimate schedule abstraction graph for workload in example 11

Proof. (proof by contradiction) Suppose there exists an execution scenario in ultimate SAG where J_2 is dispatched before J_d. According to rule 2, a causal connection must follow causal order. If J_2 has a causal connection to J_d, there exists at least one execution scenario in ultimate SAG where J_d is dispatched before J_2. If J_2 is dispatched before J_d, then the lower bound on the ultimate finish time of J_2, UFT_2^{min}, should be before the upper bound on the ultimate start time of J_d, UST_d^{max}. Given this scenario, there would be an overlap between $[\mathrm{UST}_d^{min}, \mathrm{UST}_d^{max}]$ and $[\mathrm{UFT}_2^{min}, \mathrm{UFT}_2^{max}]$. According to rule 2, if the ultimate start time of J_d has an overlap with the ultimate finish time of J_2, J_d should be causally connected to J_2, unless there exists no execution scenario where J_2 is dispatched before J_d. Since J_d does not have a causal connection to J_2, it follows that for every execution scenario in the ultimate SAG, J_2 must be dispatched after J_d. This contradicts the initial assumption, so the lemma holds. □

Based on lemma 7 and example 10, a job can be added to a causal link if it satisfies rule 3.

Rule 3. *For a causal link $\mathcal{L}$ starting from the deadline miss job J_d, if a job J_i is causally connected to the last job in $\mathcal{L}$, it is considered as a valid causal connection for $\mathcal{L}$ only if it satisfies following conditions:*

- $J_i \notin \mathcal{L}$,
- *If $J_i \Rightarrow J_d$, then $J_d \Rightarrow J_i$ (according to lemma 7).*

Algorithm 3 presents the approach for generating a causal link. The causal link is generated with an iterative depth-first approach where valid causal connection(s) are identified (based on rule 3), starting from the deadline miss job.

The input to algorithm 3 is a causal connection vector that maintains causal

connections for each job in the job set. In line 1, we initialize the link counter ($\mathrm{linkCounter}$) and the feasible solution counter ($\mathrm{solutionCounter}$) to keep track of the number of unique link explorations and the number of feasible solutions, respectively. We also maintain the energy-aware speed readjustment solution ($\mathrm{energyAwareSolution}$), which includes the energy consumption of the solution and the set of readjusted jobs along with their respective adjusted speeds. The energy consumption of $\mathrm{energyAwareSolution}$ is initialized to ∞.

To generate the causal link, $\mathcal{L}$ the valid causal connection (or one of the valid causal connections if more than one) of the last job in $\mathcal{L}$ is added to the link and the process is repeated with the last job of the causal link until the last job has no valid causal connection (lines 4-12).

When the last job of the causal link has multiple valid causal connections, to select one of them, we use branching heuristics based on the properties of the causally connected job. $\mathrm{ValidConnections}$ keeps track of valid connection(s) of jobs in the causal link. The valid connections are ordered according to the branching heuristic and the first job in the ordered set is selected to extend the causal link. To order the valid causal connections, one of the following heuristics is used

- **First-connection:** Maintain the order of valid causal connections as specified by the causal connection vector,
- **High-out:** Sort the valid causal connection job set in decreasing order based on the number of causal connections each job has, with jobs having the highest number of causal connections first,
- **Low-out:** Sort the valid causal connection job set in increasing order based on the number of causal connections each job has, with jobs having the fewest causal connections first,
- **Longest-job:** Sort the valid causal connection job set in decreasing order based on the worst-case execution time (WCET) at speed 1.0, with jobs having the longest WCET first,
- **Shortest-job:** Sort the valid causal connection job set in increasing order based on the worst-case execution time (WCET) at speed 1.0, with jobs having the shortest WCET first,

Each of the unique causal links is explored for speed readjustment (discussed in section 5.4) to resolve the deadline violation (lines 13-16). Since causal links involving the same set of jobs in different orders will result in the same speed readjustment, we consider two causal links, $\mathcal{L}_1$ and $\mathcal{L}_2$, to be unique only if $\mathcal{J}^1 \neq \mathcal{J}^2$, where $\mathcal{J}^1$ and $\mathcal{J}^2$ are jobs in $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively. The set of previously explored links is maintained in $\mathrm{exploredLinks}$.

As seen in example 7, changing the speed can change the execution scenario, resulting in a causal connection break of the explored causal link. Additionally, for a causal link that follows the correct causal order, running all jobs at the highest speed might not provide enough reduction in finish time due to limitations of finish time gains with the highest speed. If the speed readjustment does not provide a feasible solution to resolve the deadline

violation, a new causal link is generated by backtracking to the last unexplored connection and exploring the new connection until the last element in the link has no valid causal connection (lines 26-29).

As causal link exploration adds a runtime overhead to the analysis for resolving a deadline miss, we use a link exploration threshold, λ. If the number of unique explored causal links reaches λ or all unique causal links are explored without a successful speed readjustment solution, the deadline violation is resolved with a pessimistic approach discussed in section 5.5, where the speed for all causally connected jobs is readjusted to the highest speed. The link exploration threshold is used to limit the runtime overhead resulting from resolving a single deadline.

Although the first successful speed readjustment solution can be used to resolve the deadline violation, further exploration of unexplored causal links can provide a better energy-aware speed readjustment solution. Therefore, we use a feasible solution threshold, K to indicate the number of feasible solutions explored before determining an energy-aware speed readjustment solution. While a feasible solution threshold is not reached, causal link generation and exploration are continued. The energy consumption of every feasible solution is compared with the energy consumption of an existing energy-aware speed readjustment solution and the solution with the lower energy consumption is maintained as an energy-aware speed readjustment solution.

5.4. Energy-aware speed readjustment

This section describes the approaches presented for energy-aware speed readjustment using a causal link to resolve the deadline violation of a job. The objective of energy-aware readjustment is to assign a speed to each job in the causal link to resolve the deadline miss.

One of the approaches for the speed readjustment is to check different speed settings for the causal linking by iterating through the speed combinations, starting from the deadline miss job.

Example 12. Consider four jobs scheduled on two cores. When all jobs are running at the lowest speed (0.5) in their respective speed space, job J_4 violates its deadline constraint (shown in fig. 5.9(a)). As the highest priority job J_1 is released at time 0, we assume that core π_2 is claimed by J_1 till time 30.

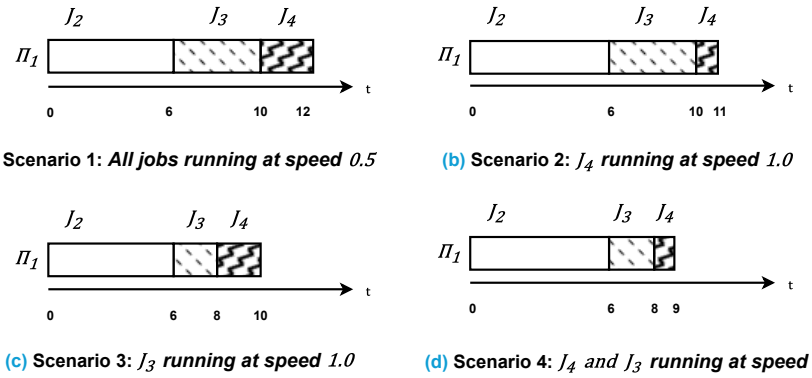
Based on rules 2 and 3, causal link is identified as $J_4 \Rightarrow J_3 \Rightarrow J_2$. To resolve the deadline violation for J_4, different speed combinations for jobs J_4, J_3. and J_2 can be explored. Starting from the first job in the link, running J_4 at a higher speed (1.0) does not resolve the deadline violation (shown in fig. 5.9(b)). As there are no other speeds to check for J_4, a speed combination is tried with connected job J_3 set to a higher speed while J_2 and J_4 are executed at the lower speed (0.5). As observed in fig. 5.9(c), this speed combination does not resolve the deadline miss. Since J_3 has no other speed, we try a speed combination with both J_3 and J_4 set to the highest speed while J_2 is executed at the lowest speed (0.5). This speed

Algorithm 3 Causal link generation algorithm

Input: Causal connection vector CCV, deadline miss job J_d, link exploration threshold, λ, and feasible solution threshold, K
Output: Casual link $\mathcal{L}$

1: $\mathrm{linkCounter} \leftarrow 0$, $\mathrm{solutionCounter} \leftarrow 0$;
2: $\mathrm{energyAwareSolution.energy} \leftarrow \infty$;
3: $\mathcal{L} \leftarrow (J_d)$;
4: $\mathrm{ValidConnections} \leftarrow ()$;
5: $\mathrm{exploredLinks} \leftarrow \{\}$;
6: Order the valid causal connection(s) of J_d from CCV according to the selected branching heuristic and add to $\mathrm{ValidConnections}$;
7: **while** $\mathrm{linkCounter} < \lambda$ and $\mathrm{solutionCounter} < K$ **do**
8: **while** $\mathrm{ValidConnections.last}$ is not empty **do**
9: $J \leftarrow$ First element removed from the end $\mathrm{ValidConnections}$
10: Add J to $\mathcal{L}$
11: Sort the valid causal connection(s) of J (according to rule 3) according to the selected branching heuristic and add to $\mathrm{ValidConnections}$;
12: **end while**
13: **if** $\mathcal{L}$ is not in $\mathrm{exploredLinks}$ **then**
14: Add $\mathcal{L}$ to $\mathrm{exploredLinks}$;
15: Increase $\mathrm{linkCounter}$ by 1;
16: $\mathrm{readjustmentSolution} \leftarrow$ Explore speed readjustment with $\mathcal{L}$;
17: **if** $\mathrm{readjustmentSolution.successful}$ **then**
18: Increase $\mathrm{solutionCounter}$ by 1;
19: $\mathrm{RE} \leftarrow \mathrm{readjustmentSolution.energy}$;
20: $\mathrm{EE} \leftarrow \mathrm{energyAwareSolution.energy}$;
21: **if** RE is lower than EE **then**
22: $\mathrm{energyAwareSolution} \leftarrow \mathrm{readjustmentSolution}$;
23: **end if**
24: **end if**
25: **end if**
26: **while** $\mathrm{ValidConnections}.last$ is empty **do**
27: Remove $\mathrm{ValidConnections}.last$;
28: Remove $\mathcal{L}.last$;
29: **end while**
30: **if** $\mathcal{L}$ is empty **then**
31: **Break**;
32: **end if**
33: **end while**

Job	Release time	Execution time at speed 1.0		Execution time at speed 0.5		Deadline	Priority
		min	max	min	max		
J_1	0	15	15	30	`30	30	0
J_2	0	3	3	6	6	10	1
J_3	0	2	2	4	4	10	2
J_4	0	1	1	2	2	9	3



(a) Scenario 1: *All jobs running at speed 0.5*

(b) Scenario 2: J_4 *running at speed 1.0*

(c) Scenario 3: J_3 *running at speed 1.0*

(d) Scenario 4: J_4 *and* J_3 *running at speed 1.0*

Figure 5.9: Speed readjustment by searching through different speed settings

combination resolves the deadline miss and provides a successful speed readjustment solution (shown in fig. 5.9(d)).

Example 12 describes the speed readjustment approach where different speed combinations are explored starting from all jobs set to the lowest speed. In scenarios, where all jobs need to be executed close to their highest speed to resolve the deadline miss, the low to high-speed combination search can result in significant exploration time. To avoid this, we use a directional search through speed combinations by determining the search direction with the speed slack. The low-speed slack, LO is defined as follows.

Definition 8. For the speed readjustment with a causal link $\mathcal{L}$ for the deadline miss job J_k, the low-speed slack is the absolute difference between the LFT_k and d_k, when all jobs in $\mathcal{L}$ are executed at the lowest speed in their respective speed setting.

In contrast, the high-speed slack, HO is defined as follows.

Definition 9. For the speed readjustment with a causal link $\mathcal{L}$ for the deadline miss job J_k, the high-speed slack is the absolute difference between the LFT_k and d_k, when all jobs in $\mathcal{L}$ are executed at the highest speed.

When $\mathrm{HO} \geq \mathrm{LO}$, we use a low-high speed combination search (similar to example 12) as the finish time of the deadline miss job is closer to its deadline when all jobs are running at lowest speed than the finish time when all jobs in the causal link $\mathcal{L}$ are executed at the highest speed. Conversely,

a high-low speed combination search is used when $\mathrm{LO} > \mathrm{HO}$, where all jobs in $\mathcal{L}$ are set to the highest speed and speed combination with reduced speeds are explored until a deadline violation.

Algorithm 4 presents the approach for directional search-based speed readjustment of a causal link $\mathcal{L}$. We first calculate HO and LO to determine the direction of the search. As discussed in section 5.3.3, the speed readjustment with a causal link with all jobs set to the highest speed can result in an unsuccessful speed readjustment due to limitations of finish time gains or causal link break from speed changes. Therefore, if the deadline violation is not resolved with all jobs in $\mathcal{L}$ running at the highest speed, then the speed readjustment is deemed unsuccessful and another causal link is generated for the speed readjustment.

If all jobs in $\mathcal{L}$ running at the highest speed can resolve the deadline violation, the direction of the search is selected based on HO and LO.

In the low-high search-based speed readjustment, each job's speed is initially set to the lowest value in its speed space, and the speed combination is explored. While not all jobs have reached the highest speed (1.0), the algorithm identifies the first job in $\mathcal{L}$ with a speed below 1.0 and increases its speed (lines 6-13). The speeds of all preceding jobs in the causal link are reset to their lowest values. If this updated speed combination leads to a successful speed readjustment, the result is returned. If not, the process continues until all jobs in $\mathcal{L}$ reach the highest speed.

For high-low search-based speed readjustment, each job's speed is initially set to the highest speed (1.0), and the speed combination is explored. While the speed combinations lead to a successful speed readjustment, the algorithm identifies the first job in $\mathcal{L}$ with speed above the lowest speed in its speed space and decreases its speed. The speeds of all preceding jobs in the causal link are reset to the highest speed, and the speed combination is explored. This process is continued until the speed combination leads to a deadline violation. After the first deadline violation, the last successful solution is returned (lines 18-38).

To limit the number of speed combination exploration of a single causal link, we use a search space threshold, ξ. When the number of explorations exceeds ξ, the result from the last exploration is returned.

The main drawback of the directional search-based speed readjustment approach is the high exploration overhead. We propose a heuristic approach based on speed slack distribution to avoid the analysis runtime overhead resulting from high exploration time. According to eq. (3.8), running shorter jobs at a higher speed incurs less energy overhead compared to longer jobs as energy is dependent on the execution time. Considering the impact of execution time on the energy, we propose a heuristic to increase (or decrease) the speed of jobs such that the combined difference in the worst-case execution time due to speed changes is close to low-speed (or high-speed) slack.

Example 13. Similar to example 12, consider four jobs scheduled on two

Algorithm 4 Directional search-based speed readjustment

Input: Causal link $\mathcal{L}$, search space threshold ξ
Output: Speed readjustment result

```
1: function lowHighSearch( )
2:     speedCombination ← the lowest speed for each job in L;
3:     readjustmentResult ← explore speedCombination;
4:     exploreCount ← 1;
5:     while ¬readjustmentResult.successful or exploreCount ≤ ξ do
6:         J_i ← the first job in L with speed below 1.0;
7:         if J_i exists then
8:             Increase the speed of job J_i;
9:             Set the speeds of all preceding jobs in L to their lowest values;
10:            readjustmentResult ← explore speedCombination;
11:            exploreCount ← exploreCount + 1;
12:        else
13:            Break;
14:        end if
15:    end while
16:    return readjustmentResult;
17: end function
18: function highLowSearch( )
19:    speedCombination ← the highest speed for each job in L;
20:    readjustmentResult ← explore speedCombination;
21:    while readjustmentResult.successful or exploreCount ≤ ξ do
22:        J_i ← the first job in L with speed above its lowest speed;
23:        if J_i exists then
24:            Decrease the speed of job J_i;
25:            Set the speeds of all preceding jobs in L to 1.0;
26:            result ← explore speedCombination;
27:            exploreCount ← exploreCount + 1;
28:            if result.successful then
29:                readjustmentResult ← result;
30:            else
31:                Break;
32:            end if
33:        else
34:            Break;
35:        end if
36:    end while
37:    return readjustmentResult;
38: end function
39: result.successful ← false ;
40: LO ← Low-speed slack of L;
41: HO ← High-speed slack of L;
42: result ← Explore speed combination with L running at 1.0;
43: if result.successful then
44:    if HO ≥ LO then
45:        result ← lowHighSearch();
46:    else
47:
48:        result ← highLowSearch();
49:    end if
50: end if
51: return result
```

cores. When all jobs are running at the lowest speed (0.5) in their respective speed space, job J_4 violates its deadline constraint. The low-speed slack for causal link $\mathcal{L}$, LO is 3. As J_4 is the shortest job, we assign a higher speed to J_4. As the speed for J_4 is set to 1.0 and the change in WCET is less than LO (as shown in fig. 5.9(b)), we increase the speed of the next shortest job (based on WCET at speed 1.0), J_3. Since the combined change in WCET ($1+2$) resulting from changing J_4 and J_3 is greater than or equal to LO, the slack distribution is complete and the updated speed combination is explored. The resulting execution scenario is the same as fig. 5.9(d).

As seen in example 13, a successful readjustment is achieved with just one exploration, compared to three explorations for example 12.

Algorithm 5 presents the proposed solution for speed slack distribution-based speed readjustment. For low-speed slack distribution, all job speeds are set to the lowest speed in their speed space and the speed of the shortest job (based on WCET at speed 1.0) is increased until the combined difference in the worst-case execution time due to the speed increase is greater than or equal to the low-speed slack (lines 3-17). Although the exploration time of casual link is reduced with speed slack distribution-based speed readjustment, for low-speed slack distribution of a $\mathcal{L}$, the speed combination result is fixed as the approach is deterministic. Therefore, if a speed combination with a low-speed slack distribution provides unsuccessful speed readjustment (due to a causal link break), a speed combination is generated based on a high-speed slack distribution. If speed readjustment is successful for all jobs in the causal link running at the speed 1.0, then the high-speed slack is identified. Similar to the search-based approach, if all jobs in the causal link running at the speed 1.0 result in an unsuccessful speed readjustment, then a new causal link is generated.

For high-speed slack distribution, all jobs are set to speed 1.0 and the speed of the longest job (based on WCET at speed 1.0) is reduced (similar to example 13) until the combined difference in the worst-case execution time due to the speed decrease is less than the high-speed slack (lines 19-36). Compared to the search-based speed readjustment approach, speed slack distribution-based speed readjustment has a lower exploration time overhead due to limited speed combination exploration (max three).

After a successful speed readjustment is complete, the speed space for the readjusted jobs is updated to exclude any speeds lower than the readjusted speed. Additionally, the schedule abstraction graph (from section 5.2) is reverted to its state before any of the readjusted jobs are dispatched, and the SAG exploration resumes from there.

The ultimate SAG (from section 5.3.1)) is also reverted to its state before any of the readjusted jobs are dispatched. As a result, the exploration of ultimate SAG (for a potential future deadline violation) is restricted to the execution scenarios that are possible with the updated speed space of readjusted jobs. This means that any execution scenarios arising from the removed speed are no longer explored. Additionally, the ultimate start and finish time bounds of the readjusted jobs are reset to their initial values ($[\infty, 0]$).

Algorithm 5 speed slack distribution-based speed readjustment

Input: Causal link $\mathcal{L}$
Output: Speed readjustment result

```
1: function getSpeedCombination(slack)
2:     speedCombination ← ()
3:     if slack is LO then
4:         speedCombination ← lowest speed for each job in L;
5:         idx ← indices of L sorted by increasing WCET at speed 1.0;
6:         reducedWCET ← 0;
7:         while reducedWCET < slack do
8:             id ← idx.first
9:             if speedCombination[id] is 1.0 then
10:                Remove idx.first from idx;
11:            else
12:                oldET ← WCET of L[id] at speed speedCombination[id];
13:                Increase the speed of speedCombination[id] to the next
                   higher speed in Z_id;
14:                newET ← WCET of L[id] at speed speedCombination[id];
15:                reducedWCET ← reducedWCET + oldET − newET;
16:            end if
17:        end while
18:    else
19:        speedCombination ← speed 1.0 for each job in L;
20:        idx ← indices of L sorted by decreasing WCET at speed 1.0;
21:        increasedWCET ← 0;
22:        while idx is not empty do
23:            id ← idx.first
24:            if speedCombination[id] is lowest speed in Z_id then
25:                Remove idx.first from idx;
26:            else
27:                oldET ← WCET of L[id] at speed speedCombination[id];
28:                Decrease the speed of speedCombination[id] to the next
                   lower speed in Z_id;
29:                newET ← WCET of L[id] at speed speedCombination[id];
30:                increasedWCET ← increasedWCET + oldET − newET;
31:                if increasedWCET > slack then
32:                    Increase the speed of speedCombination[id] to the next
                       higher speed in Z_id;
33:                    Remove idx.first from idx;
34:                end if
35:            end if
36:        end while
37:    end if
38:    return speedCombination;
39: end function
40: result.successful ← false ;
41: LO ← Low-speed slack of L;
42: LSCombination ← getSpeedCombination( LO );
43: result ← explore LSCombination;
44: if ¬result.successful then
45:    speedCombination ← the highest speed for each job in L;
46:    result ← explore speedCombination;
47:    if result.successful then
48:        HO ← High-speed slack of L;
49:        HSCombination ←getSpeedCombination( LO );
50:        result ← explore HSCombination;
51:    end if
52: end if
53: return result;
```

5.5. Timeout

The speed readjustment has added time for exploring different speed combinations on each unique causal link. In certain scenarios, exploration time can be significantly high, particularly when there are many deadline misses to address or a high number of unique causal links without successful speed readjustment. Additionally, the search threshold (discussed in section 5.4) and link exploration threshold (discussed in section 5.3.3) limit the causal link exploration.

To manage the runtime of the energy-aware speed assignment analysis, a timeout threshold is implemented. When analysis runtime exceeds this threshold or if causal link exploration is pruned due to search and link exploration thresholds, speed readjustment for a deadline violation is performed by setting the causally connected job set to the highest speed, without exploring individual unique causal links. We define causally connected job set as follows

Definition 10. Causally connected job set for a deadline miss job J_d, denoted by CC_d, is a set of unique jobs from all possible causal links starting from J_d.

Algorithm 6 Causally connected job set generation

Input: Causal connection vector CCV, deadline miss job J_d
Output: Causally connected job set CC_d

1: $\mathrm{ES} \leftarrow \{J_d\}$
2: $\mathrm{CC}_d \leftarrow \{\}$
3: **while** ES is not empty **do**
4: $J \leftarrow$ Job removed from ES
5: connections $\leftarrow \mathrm{CCV}[J]$
6: **for** each job J' in connections **do**
7: **if** $J' \notin \mathrm{CC}_d$ **then**
8: Add J' to CC_d
9: Add J' to ES
10: **end if**
11: **end for**
12: **end while**
13: **return** CC_d

Algorithm 6 presents the approach for generating the causally connected job set CC_d for a deadline miss job J_d through an iterative exploration of all unique connections starting from J_d. One of the inputs to the algorithm is the causal connection vector, CCV that consists of causal connections for each job in $\mathcal{J}$.

In line 1, explore set, ES is initialized with the deadline miss job, J_d. The set ES maintains all jobs whose connections have yet to be explored. Simultaneously, an empty set CC_d is initialized at line 2 to hold the causally connected job set.

In each iteration (lines 3-12), a job from ES is removed, and its causal con-

nections are identified from CCV. Every causally connected job that is not recorded in CC_d is added to CC_d. Additionally, this job is also added to ES for further exploration of its connections. This process continues until ES is empty, meaning that all connected jobs have been fully explored and no unexplored connections remain.

Lemma 8 shows that if the job set is schedulable when all jobs are executed at the highest speed, then running the causally connected job set for the deadline-violating job J_d at the highest speed will resolve the deadline miss for J_d.

Lemma 8. *Consider a job set $\mathcal{J}$ that is schedulable at the highest speed $\mathcal{S}^{max} = 1.0$ on a given system. If running each job $J_i \in \mathcal{J}$ at $\min\{S|S \in Z_i\}$ results in a deadline miss at job J_d, then running all jobs in CC_d at $\mathcal{S}^{max}$ will certainly resolve the deadline violation of J_d.*

Proof. Assume that the job set $\mathcal{J}$ is schedulable at the highest speed $\mathcal{S}^{max}$ on a given system. According to the definition 10, the causally connected job set CC_d consists of jobs from all possible causal links. These causal connections are identified based on all possible speed combinations. Therefore, the causal connections from J_d when all jobs are executed at the highest speed are also considered.

This ensures that any job influencing the finish time of job J_d when all jobs are executed at the highest speed is included in CC_d. If $\mathcal{J}$ is schedulable at the highest speed $\mathcal{S}^{max}$, then running CC_d at the highest speed creates the execution scenario similar to all jobs running at the highest speed, solving the deadline violation of J_d. □

The timeout approach prioritizes analysis runtime over energy reduction by reducing the exploration time for speed readjustment. It pessimistically assigns all connected job sets to the highest speed for each detected deadline violation, to avoid exploring individual causal links.

Similar to the speed readjustment approaches discussed in section 5.4, the speed space for each readjusted job and SAG state space is updated. After speed readjustment, energy-aware speed assignment analysis continues with all jobs running at the lowest speed in the speed space until the analysis is complete.

6

Empirical evaluation

We conducted experiments to answer the following questions: (i) How much energy reduction is achieved with the proposed solution? (ii) What is the analysis runtime overhead of the energy-aware speed assignment compared to the schedule abstraction graph-based schedulability analysis? (iii) How do energy-aware speeds impact the schedulability of the job set? (iv) How do the job set parameters impact the obtained energy reduction and analysis runtime overhead? (v) How do the job set parameters impact the obtained energy reduction and analysis runtime overhead?

6.1. Experimental setup

To answer these questions, we conducted experiments on the proposed solution with a wide range of synthetic task sets. This section discusses the experimental setup used for these experiments.

Baselines: To evaluate the impact on energy reduction and runtime overhead of energy-aware speed assignment, we compared two variations of the energy-aware speed assignment approach (as discussed in fig. 5.1) with every job executed at the highest speed (referred to as 'Single speed analysis'). Since no directly comparable work exists, this evaluation highlights the performance of our approach with a schedule abstraction graph-based schedulability analysis framework. The first variation uses the directional search-based speed readjustment approach (as presented in algorithm 4), while the second variation uses the speed slack distribution-based speed readjustment (as explained in algorithm 5). Additionally, we also evaluate the impact of pessimistic timeout speed readjustment (referred to as All-connected-high speed readjustment) (as discussed in section 5.5) on energy reduction and analysis runtime.

Since the speed slack distribution-based speed readjustment is a heuristic approach, we anticipate a lower runtime overhead with comparable energy reduction compared to the directional search-based speed readjustment. The energy reduction and analysis runtime overhead of the pessimistic time-

out speed readjustment is expected to be lower than both variations, as it avoids exploring individual causal links. Given that all approaches use schedulability as a criterion for successful speed readjustment, the schedulability is expected to match that of the job set running at the highest speed.

Evaluation platform: The experiments were performed using the DelftBlue supercomputer at Delft High-Performance Computing Center [43], which hosts Compute nodes with Intel Xeon E5-6248R 24C 3.0GHz CPUs and 192 GB of Memory per node. The analysis was implemented as a single-threaded C++ program.

Metrics: We use the following metrics to evaluate the performance of the presented energy-aware speed assignment approach.

- **Energy reduction:** As the goal of energy-aware speed assignment is to reduce the energy consumption with static slack reclamation, the key metric for determining the effectiveness of the proposed solution is energy reduction compared to the energy consumption with the speed assignment where the highest speed is assigned to all jobs. The energy consumption of the job set for a given speed assignment is calculated as the sum of the energy consumption of each job in the job set with the assigned speed, i.e., the energy consumption due to the idle core is not considered.

  The energy reduction percentage is calculated as

  $$\text{Energy reduction (\%)} = (1 - \frac{\mathcal{E}_{energyAware}}{\mathcal{E}_{high}}) \times 100,$$

  where $\mathcal{E}_{energyAware}$ is the energy consumption of the job set with the energy-aware speed assignment, while $\mathcal{E}_{high}$ is the energy consumption of the job set with all jobs running at the highest speed. The higher number of energy reduction indicates better performance of the energy-aware speed assignment.
- **Analysis runtime overhead:** As the proposed approach performs energy-aware speed readjustment in addition to the schedulability analysis with a schedule abstraction graph, we evaluate the analysis runtime overhead of the energy-aware speed assignment compared to the runtime of schedule abstraction graph schedulability analysis. The Analysis runtime overhead is a ratio of the total runtime of the energy-aware speed assignment (including the schedulability analysis) to the runtime of the schedulability analysis with the schedule abstraction graph when all jobs are executed at the highest speed. Analysis runtime overhead is calculated as

  $$\text{Analysis runtime overhead} = \frac{R_{energyAware}}{R_{high}},$$

  where $R_{energyAware}$ is the total runtime of the energy-aware speed assignment (including the schedulability analysis with SAG, as proposed in fig. 5.1), while R_{high} is the runtime of the schedulability analysis with the SAG when all jobs are executed at the highest speed. The lower

value of analysis runtime overhead indicates better performance of the energy-aware speed assignment.

- **Schedulability ratio:** Since we consider a hard real-time system, the objective of energy-aware speed assignment is to reduce the energy consumption of the job set without violating the deadline constraints of the job. The schedulability ratio is calculated as the ratio of job sets deemed to be schedulable divided by the number of considered job sets. For the schedulability analysis with all jobs executing at the highest speed, the job set was claimed unschedulable if an execution scenario with a deadline miss was found. For the schedulability analysis with the energy-aware speed assignment, the job set was considered to be unschedulable if the energy-aware speed readjustment was unsuccessful for a deadline miss.

Energy and speed model: The power and speed model is based on the model by Park et. al. [38]. The authors considered a system with a Samsung Exynos 4210 processor (based on the ARM Cortex-A9 core) which is commonly used for embedded system applications. They obtained values for the coefficients C_{ef}, α_1, and α_2 by performing a power characterization procedure by analyzing the measured load current of an active processor with different frequencies f_i (and required voltages for the frequencies V_{dd_i}). The values for V_{dd_i} and f_i are provided in table 6.1, where speed $\mathcal{S}_i$ is the ratio of the selected frequency with the highest frequency (As mentioned in the system model).

According to the power characterization by park et. al. [38], C_{ef} is 0.446, α_1 is 0.1793, and α_2 is set to -0.1527. The energy consumption of a job is calculated by eq. (3.8), where the power consumption of an active processor at speed $\mathcal{S}$ is given as (from park et. al. [38] and based on eq. (3.7))

$$P(\mathcal{S}) = 0.446.{V_{dd_i}}^2.f_i + 0.1793.V_{dd_i} - 0.1527.$$

Table 6.1: Voltage $V_{dd_i}(V)$ and clock frequency $f_i(GHz)$ levels for Samsung Exynos 4210 (from Park et. al. [38])

DVFS level	$\mathbf{V_{dd_i}}$	$\mathbf{f_i}$	$\mathcal{S}_\mathbf{i}$
Level 1	1.2	1.4	1.0
Level 2	1.15	1.3122	0.94
Level 3	1.1	1.2218	0.87
Level 4	1.05	1.12870	0.8
Level 5	1.0	1.0327	0.74

The resulting speed range for this experiment is $\{0.74, 0.80, 0.87, 0.94, 1.00\}$, where 0.74 is $\mathcal{S}^{min}$. As we consider speed range from $\{\mathcal{S}^{crtl}, \ldots, \mathcal{S}^{max}\}$, any speed $\mathcal{S}_i \in \{\mathcal{S}^{min}, \ldots, \mathcal{S}^{max}\}$ is removed in preprocessing if $\mathcal{S}_i < \mathcal{S}^{crtl}$, i.e., energy consumption of speed $\mathcal{S}_i$ is higher than next speed in discrete speed range($\mathcal{E}(\mathcal{S}_i) > \mathcal{E}(\mathcal{S}_{i+1})$).

Job set generation: We generated 100 synthetic periodic task sets for each data point using the technique described by Emberson et al. [44]. For each

task set, we created a job set (as discussed in section 3.1.1) with priority assignments based on the earliest deadline first (G-NP-EDF) scheduling policy.

To randomly generate a periodic task set with n tasks and a given utilization U, we first randomly generated n period values in the range $[10, 100]ms$ with a granularity of $5ms$ using a log-uniform distribution (Similar to Nasri et. al. [1], [15]). Next, we generated n task-utilization values that sum to the expected utilization U using the RandFixedSum algorithm [45]. Based on the periods and task utilization values, we calculated the worst-case execution time (WCET), C_i^{max} for each task with best-case execution time (BCET), C_i^{min}, set to $0.6 \times C_i^{max}$.

When generating the job set, the release jitter σ is set to 100 microseconds. To effectively evaluate the impact of energy-aware speed assignment, we excluded any task sets that were trivially unschedulable, i.e., the task sets that were unschedulable even without execution time variation or release time jitter (note that this initial test serves as a necessary schedulability test but is not sufficient due to the possibility for schedulability anomalies under non-preemptive conditions). We also excluded any task sets with more than $100,000$ jobs per hyperperiod.

For energy-aware speed assignment analysis, we set the timeout threshold (as discussed in section 5.5) to 9000 seconds. Once the analysis runtime exceeds this threshold, all deadline misses are addressed using the timeout speed readjustment approach. As default analysis parameters, We used first-connection branching for causal link exploration with the link exploration threshold λ set to 50 and the feasible solution threshold K set to 1. For the directional search-based speed readjustment, the default search space threshold ξ was set to 100.

6.2. Impact of system utilization

The goal of this experiment is to evaluate the impact of increasing total utilization on energy-aware speed assignment analysis. We considered a system with four cores ($m = 4$) and six tasks ($n = 6$). The total system utilization U was chosen from $10 - 70\%$ utilization (with $m = 4$).

Figures 6.1 and 6.2 illustrate the impact of increasing utilization on different speed readjustment approaches (discussed in sections 5.4 and 5.5). As expected, the energy reduction declines with increasing utilization U since increasing total utilization decreases the static slack available for energy reduction. Figure 6.1(a) describes the schedulability ratio of the job sets. The schedulability of job sets with energy-aware speed assigned with different speed readjustment approaches is consistent with the baseline. This

Table 6.2: The number of schedulable job sets in fig. 6.2(a) that required speed readjustment over increasing utilization.

Utilization (%)	10%	20%	30%	40%	50%	60%	70%
Count	0	3	25	52	41	29	5

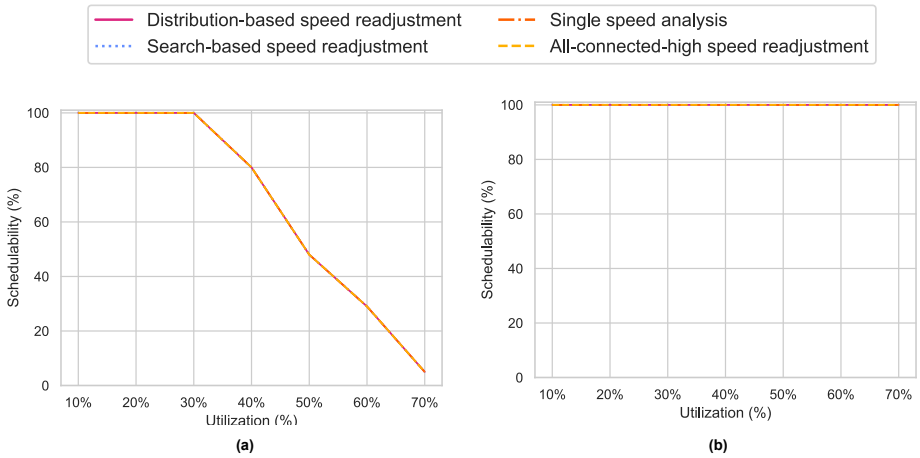


Figure 6.1: Impact of utilization on schedulability ratio following a (a) necessary test, (b) sufficient test.

indicates that every job set that is deemed schedulable by the schedule abstraction graph remains schedulable when job execution time intervals are based on the assigned energy-aware speed.

Figure 6.2(a) shows the impact of increasing utilization on energy reduction of schedulable task sets. As seen in table 6.2, all job sets at 10% utilization are trivially schedulable with the lowest speed in the speed setting, without any speed readjustments. When all jobs are running at the lowest speed 0.74 is close to 32%, which is the upper bound on energy reduction with the speed range selected for the experiments.

To evaluate the impact of different speed readjustment approaches, figs. 6.2(c) and 6.2(d) shows the effect of increasing utilization on the schedulable job sets that required speed readjustment. As seen in fig. 6.2(d), analysis runtime overhead for schedulable job set increases with increasing utilization as the number of speed readjustments in the energy-aware speed assignment analysis also increases. Energy reduction and analysis runtime overhead for all speed readjustment approaches are similar up to 30% utilization, as limited speed adjustments are needed to make the job sets schedulable at this level.

From 40% utilization, the highest energy reduction is observed with the search-based speed readjustment approach. This method conducts a directional search through multiple speed combinations of a causal link to resolve deadline misses, resulting in significantly high runtime overhead. The distribution-based approach achieves slightly lower energy reduction than the search-based approach, as it uses a heuristic to assign speeds to jobs in a causal link. However, the distribution-based method offers significantly lower runtime overhead than the search-based approach. As expected, the All-connected-high approach results in lower energy reduction and runtime overhead, as it assigns the highest speed to all connected jobs, unlike other approaches that adjust speeds for individual causal links.

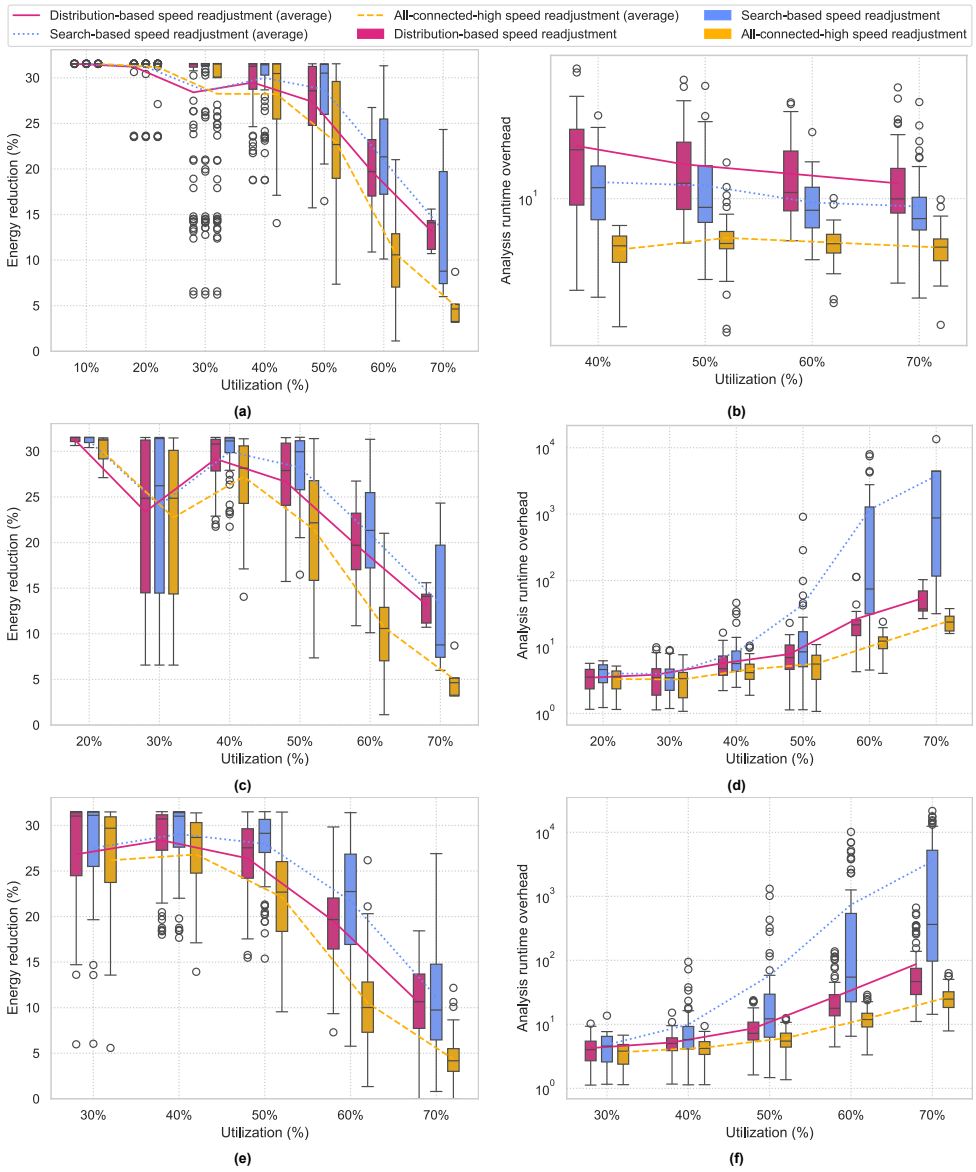


Figure 6.2: Impact of utilization on (a,c,e) energy reduction (with schedulable job sets, speed readjusted job sets, and speed readjusted job sets with 100% schedulability, respectively) and (b,d,f) analysis runtime overhead (unschedulable job sets, schedulable speed readjusted job sets, and schedulable speed readjusted job sets with 100% schedulability, respectively) for different speed readjustment approaches.

The analysis runtime overhead for unschedulable job set is shown in fig. 6.2(b). These job sets were found to be unschedulable both when all jobs were running at the highest speed and when using energy-aware speed assignment analysis. When a job set is unschedulable, the distribution-based approach incurs more runtime overhead than the search-based approach as the search-based approach executes the causal links at the highest speed to

determine the search direction and only starts the speed combination search if the highest speed solves the deadline miss. In contrast, the distribution-based approach initially explores low-speed slack distributions before checking if the causal link at the highest speed solves the deadline miss. When the deadline miss cannot be solved, this additional exploration for each causal link leads to higher runtime overhead.

As seen in the fig. 6.1(a) and table 6.2, only five schedulable job sets were evaluated at 70%. To mitigate any bias from the schedulability ratio at high utilization, we repeated the experiment with 100 job sets that are schedulable at the highest speed (the job sets follow the sufficient schedulability test). The results exhibit a similar trend as observed in figs. 6.1(b), 6.2(e) and 6.2(f).

It is worth noting that the counter-intuitive drop in energy reduction at 30% utilization is due to a skewed task utilization distribution. At this utilization, a single task is assigned a very high utilization (over 22.5%), while the remaining tasks contribute to less than 7.5% utilization. This imbalance observed at 30% utilization, in several outliers of fig. 6.2(a), limits energy reduction because the high-utilization task must run at a higher speed due to low slack, increasing energy consumption. The impact of a high-utilization task on energy reduction is significant if the high-utilization task has a considerably short period, causing frequent blocking of other tasks, or a considerably long period, resulting in prolonged high-speed execution.

6.3. Impact of causal link branching heuristic

For energy-aware speed readjustment, causal links are generated using a depth-first approach, with a branching heuristic (discussed in section 5.3.3) guiding the connection selection. To understand the impact of branching heuristics on energy-aware speed assignment analysis, we considered a system with four cores ($m = 4$) and six tasks ($n = 6$), with the total system utilization set to $U = 40\%$. The branching heuristic for the analysis is selected from five heuristics presented in section 5.3.3.

As the branching heuristics are used to generate the causal link through a depth-first approach, the effectiveness of these heuristics is expected to influence the causal link exploration overhead. As observed in fig. 6.3(a), the choice of branching heuristic has minimal impact on energy reduction, with a slight increase observed for distribution-based with the longest-job branching heuristic. The longest-job branching heuristic selects the job with the highest worst-case execution time when generating the causal link. Since the distribution-based approach initially performs low-speed slack distribution by increasing the speed of the smallest job in the link, the slack distribution is more likely to be successful with longer jobs. This results in fewer cases of high-speed slack distribution, leading to a marginally higher energy reduction.

Regarding analysis runtime overhead, the impact of branching heuristics is minimal with increased analysis runtime overhead for high-out heuristics with search-based speed readjustment. This increase in average runtime

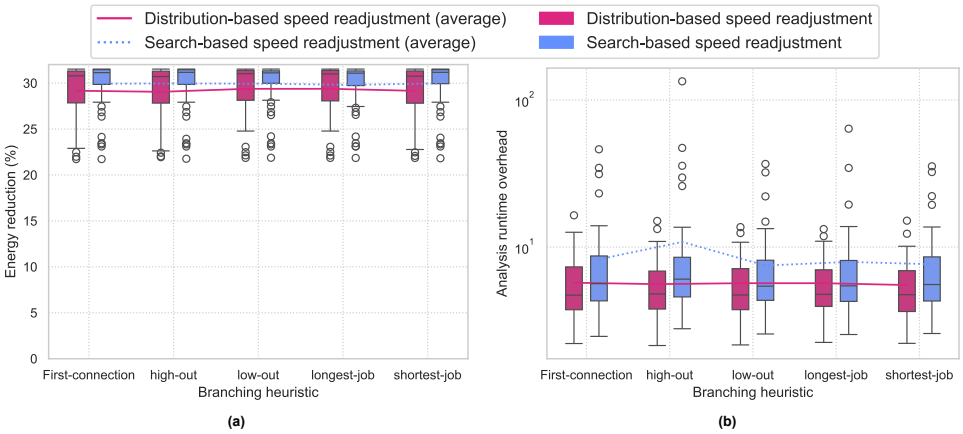


Figure 6.3: Impact of branching heuristics on (a) energy reduction and (b) analysis runtime overhead with various speed readjustment methods.

overhead is attributed to outliers, as the median and interquartile range of the box plot align with other heuristics. High-out heuristics select jobs based on their number of causal connections, which results in longer causal links. The exploration time for a causal link in search-based speed readjustment is proportional to the number of jobs within the link, meaning longer causal links incur higher runtime overhead.

6.4. Impact of search space threshold

In directional search-based speed readjustment, the search space for each causal link is bounded by the search space threshold. This experiment analyzes the impact of search space threshold ξ on energy-aware speed assignment analysis for schedulable job sets that require speed readjustment. We considered a system with four cores ($m = 4$), six tasks ($n = 6$), and $U = 40\%$. The search space threshold ξ was varied from the range 10 to 150 in increments of 10. As observed in fig. 6.4(a), the search space threshold has minimal impact on energy reduction with average energy reduction improved by just 1% at $\xi = 40$ compared to $\xi = 10$. This minor increase in average energy reduction is attributed to the edge case scenarios when restricted search space causes causal link exploration to fail. When the search space threshold is raised, fewer causal links are pruned, leading to a modest improvement in energy reduction for these outliers.

This trend is evident with the outliers in fig. 6.4(b). For a small value of ξ, runtime overhead is high as many causal links are explored and pruned due to a lower search space threshold. As ξ increases, fewer causal links are explored since a feasible solution is found with increased search space, as seen at $\xi = 40$. However, beyond $\xi = 40$, the runtime overhead grows again as exploring each causal link takes longer. Notably, the search space threshold has a very low impact on median values for runtime overhead, as ξ primarily affects the scenarios when the feasible solution for speed setting is considerably far from the lowest speed for multiple causal links.

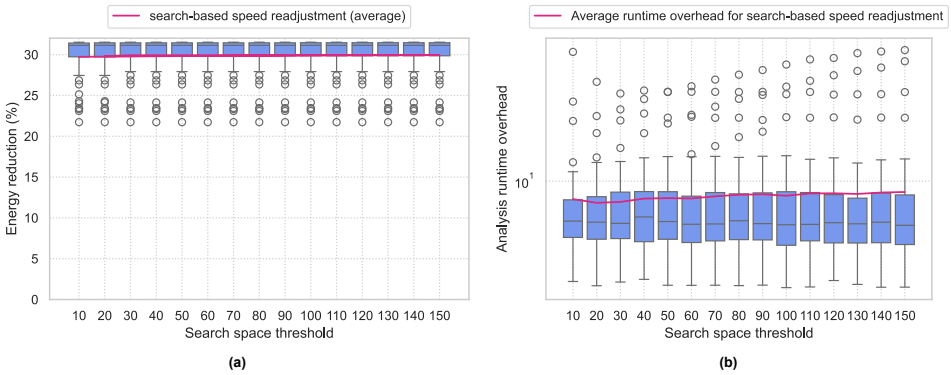


Figure 6.4: Impact of search space threshold ξ on (a) energy reduction and (b) analysis runtime overhead with search-based speed readjustment.

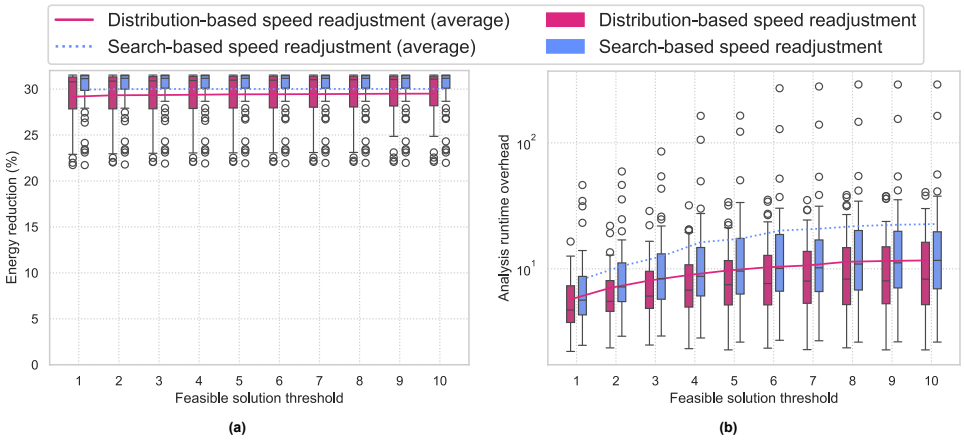


Figure 6.5: Impact of feasible solution threshold K on (a) energy reduction and (b) analysis runtime overhead with various speed readjustment methods.

6.5. Impact of feasible solution threshold

This section discusses the effect of exploring multiple feasible solutions for a single deadline violation in schedulable job sets that require speed readjustment. The threshold for the number of speed readjustment solutions explored for each deadline violation is defined by K. With an increasing value of K, more feasible solutions are explored before deciding on an energy-aware solution. We considered a system with four cores ($m = 4$), six tasks ($n = 6$), and $U = 40\%$, with the feasible solution K varied from 1 to 10. As seen in fig. 6.5(a), increasing the value of K results in a slight improvement in energy reduction for distribution-based speed readjustment. However, with an increasing value of K, a significant runtime overhead is incurred for both speed readjustment approaches as exploration time for each deadline violation is considerably increased. This can be observed in fig. 6.5(b).

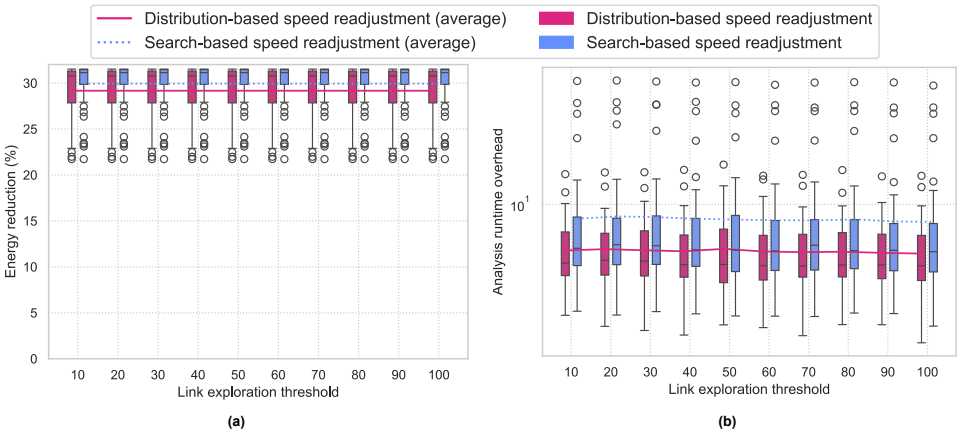


Figure 6.6: Impact of link exploration threshold λ on (a) energy reduction and (b) analysis runtime overhead with various speed readjustment methods.

6.6. Impact of causal link exploration threshold

In energy-aware speed readjustment, causal links are generated to reduce the search space for speed readjustment. To investigate the impact of the number of causal links explored on energy-aware speed assignment, we considered a system with four cores ($m = 4$), six tasks ($n = 6$), 40% utilization ($U = 40\%$), and varied link exploration threshold λ from 10 to 100 in increments of 10.

As seen in figs. 6.6(a) and 6.6(b), λ has no impact on energy reduction or runtime overhead, as the number of unique causal link explorations rarely exceed λ threshold. Since $K = 1$, the speed readjustment for a deadline violation is completed when a single feasible solution is found. Even for K threshold till 10 (from fig. 6.5), the maximum number of unique causal link exploration for a deadline violation speed readjustment was 32, which did not exceed the causal link exploration threshold as λ was set to 50. Although we expect to see a minor decline in the runtime overhead and energy reduction of outliers with very low values of λ.

6.7. Impact of the number of tasks

This section discusses the impact of the number of tasks on energy-aware speed assignments with different speed readjustment approaches by analyzing schedulable job sets that require speed readjustment. For $m = 4$ and $U = 40\%$, we selected the number of tasks n per task set from $\{5, 6, 7, 8, 9, 10, 11, 12\}$.

As observed in fig. 6.7(a), the impact of increasing the number of tasks with constant utilization shows no significant trend in energy reduction. However, there is a noticeable drop in energy reduction at $n = 5$ due to skewed task utilization distribution, similar to the energy reduction drop observed at $U = 30\%$ in fig. 6.2(a).

Considering the analysis runtime overhead for schedulable job sets that re-

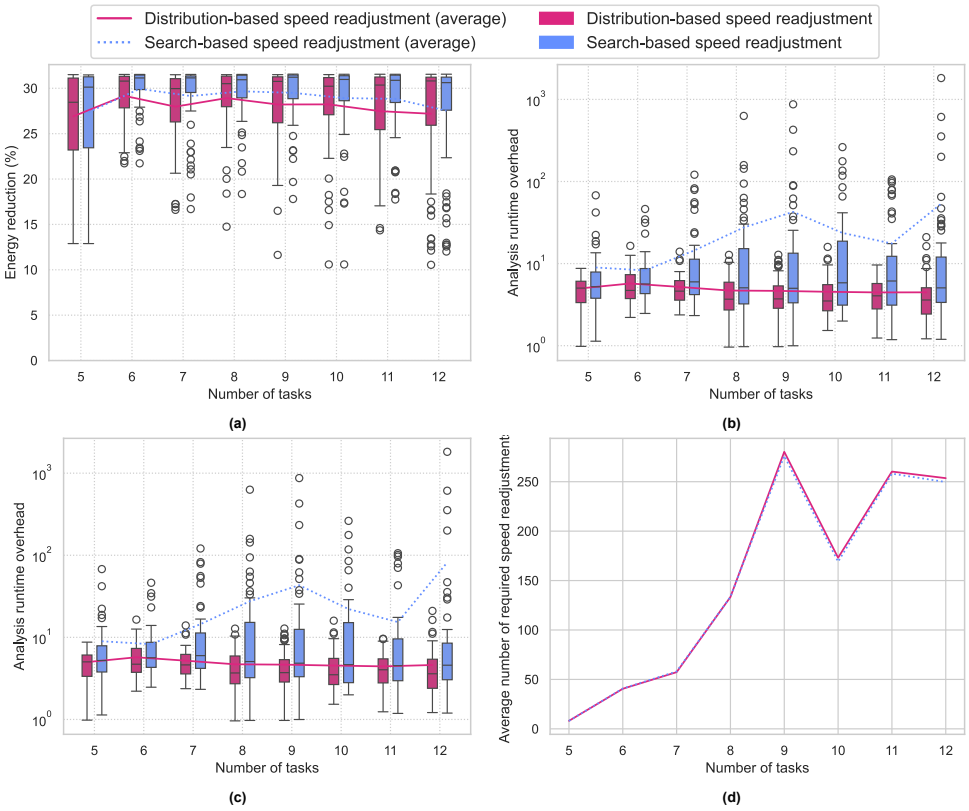


Figure 6.7: Impact of the number of tasks (a) energy reduction, (b,c) analysis runtime overhead (with and without timeout job sets, respectively), and (d) speed readjustment count with various speed readjustment methods

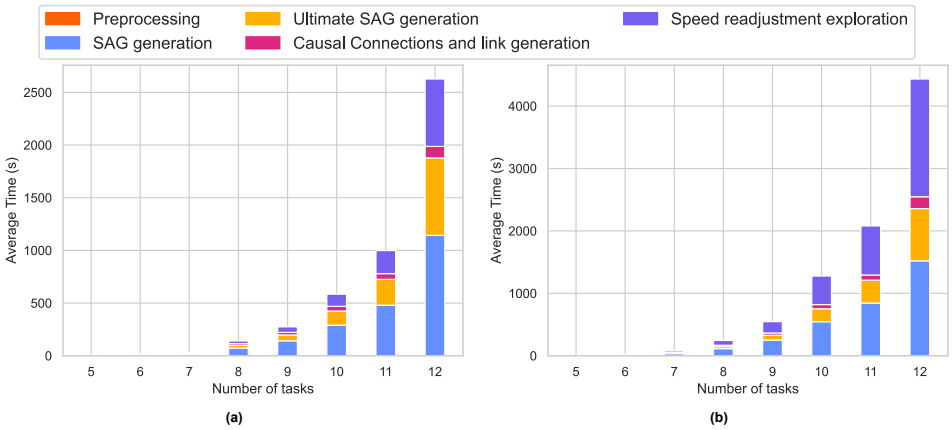


Figure 6.8: Impact of the number of tasks on analysis runtime of (a) distribution-based and (b) search-based speed readjustment approach with the proportion of time spent on each step of energy-aware speed assignment

quired speed readjustment, fig. 6.7(b) demonstrates a higher runtime overhead for search-based speed readjustment compared to the distribution-

based approach, due to the low exploration overhead of the distribution-based approach. Additionally, as the number of tasks increases with constant utilization, there is a corresponding increase in the number of jobs and execution scenarios, leading to higher analysis runtime and causal connection amongst jobs. This trend can be observed in fig. 6.9(b), where the number of jobs resulting in timeouts increases with the number of tasks. As discussed in section 5.5, when the energy-aware speed assignment analysis runtime exceeds the timeout threshold (9000), any subsequent deadline miss is resolved by assigning the causally connected job set of deadline miss job to the highest speed. Figure 6.7(c) shows that even after excluding the runtime overhead for all job sets that exceed the timeout threshold, the overall pattern of runtime overhead across an increasing number of tasks remains consistent. As observed in fig. 6.7(d), the average runtime overhead of the search-based speed readjustment approach is attributed to the average required speed readjustment and high exploration overhead.

Figure 6.8(a) shows that the average analysis runtime of distribution-based speed readjustment increases with increased number of tasks. In contrast,fig. 6.8(b) reveals that the search-based approach leads to a significant increase in analysis runtime, with a substantial portion of the time dedicated to speed readjustment exploration. This exploration time is notably higher compared to the distribution-based method. Unlike distribution-based speed readjustment, the proportion of speed readjustment exploration for the search-based approach slightly increases with an increased number of tasks, largely due to the added overhead from the increase in causal connections.

Although ultimate SAG generation is generally expected to have a higher runtime than SAG due to high execution time uncertainty, figs. 6.8(a) and 6.8(b) show that ultimate SAG has a lower overhead than SAG generation. This behaviour can be attributed to the fact that the ultimate execution time bound is recalculated after each speed readjustment, and the ultimate SAG is only explored until the last deadline miss is detected. Additionally, the ultimate SAG approach may offer a greater potential for state merging, which can reduce the number of states that need to be explored, ultimately leading to a lower runtime.

Finally, fig. 6.9(a) shows the schedulability ratio as the number of tasks increases, which remains consistent across both variations of the energy-aware speed assignment approach.

6.8. Impact of number of cores

We evaluate the effect of increasing the number of cores on energy-aware speed assignment analysis by analyzing schedulable job sets that require speed readjustment. we vary the number of cores m from 2 to 8 with the number of tasks n configured as $1.5 \times m$, and the utilization U was set 40%, maintaining a constant core-to-task and core-to-utilization ratio.

Similar to section 6.7, no significant trend in energy reduction was observed in fig. 6.10(a) as the utilization is fixed. However, as shown in fig. 6.10(b),

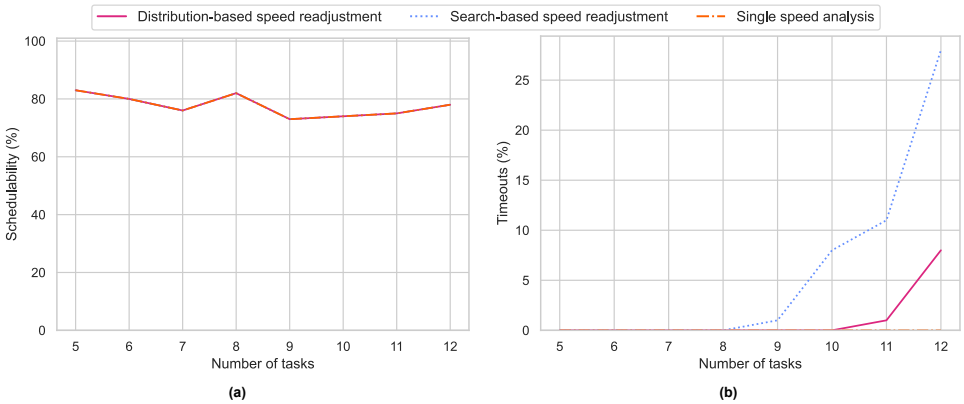


Figure 6.9: Impact of the number of tasks on (a) schedulability and (b) timeout count with various speed readjustment methods.

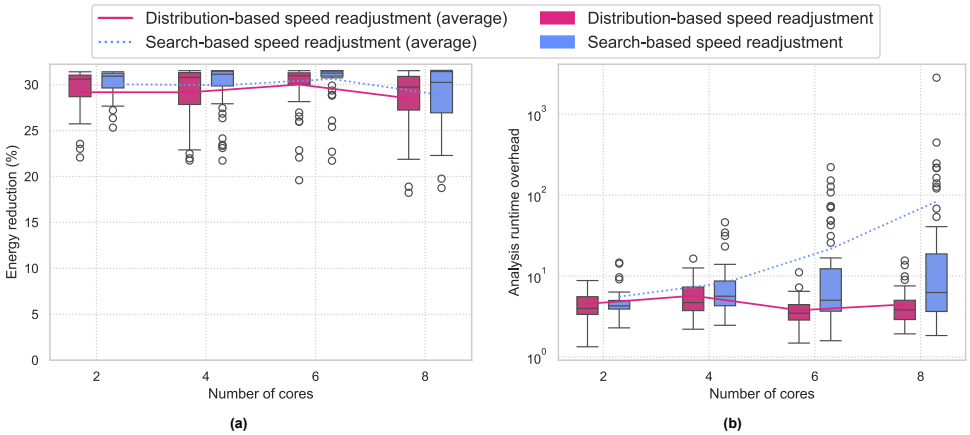


Figure 6.10: Impact of the number of cores (a) energy reduction and (b) analysis runtime overhead with various speed readjustment methods.

the analysis runtime overhead of the search-based approach significantly increases with the number of cores, as a high number of cores leads to more execution scenarios and causal connections. Since the core-to-task ratio and core-to-utilization ratio are fixed, the impact of an increasing number of tasks is balanced. As seen in fig. 6.11(b), the increase in the number of cores results in more timeout scenarios due to higher analysis runtime overhead. Finally, fig. 6.11(a) demonstrates that, over varying the number of cores, the schedulability ratio of the job sets remains consistent with the baseline where all of the jobs are assigned to the highest speed.

6.9. Impact of execution time variation

This section discusses the impact of execution time variation on energy-aware speed assignment using different speed readjustment approaches by analyzing schedulable job sets that require speed readjustment. We consider a system with four cores ($m = 4$), six tasks ($n = 6$), 40% utilization

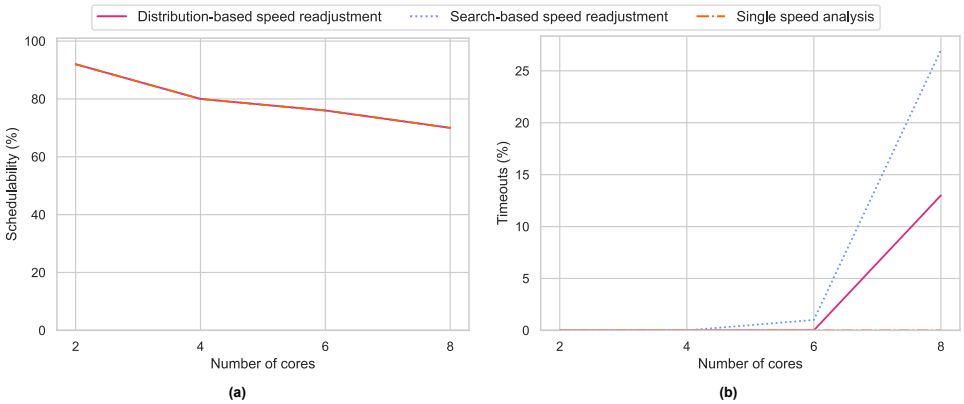


Figure 6.11: Impact of the number of cores on (a) schedulability and (b) timeout count with various speed readjustment methods.

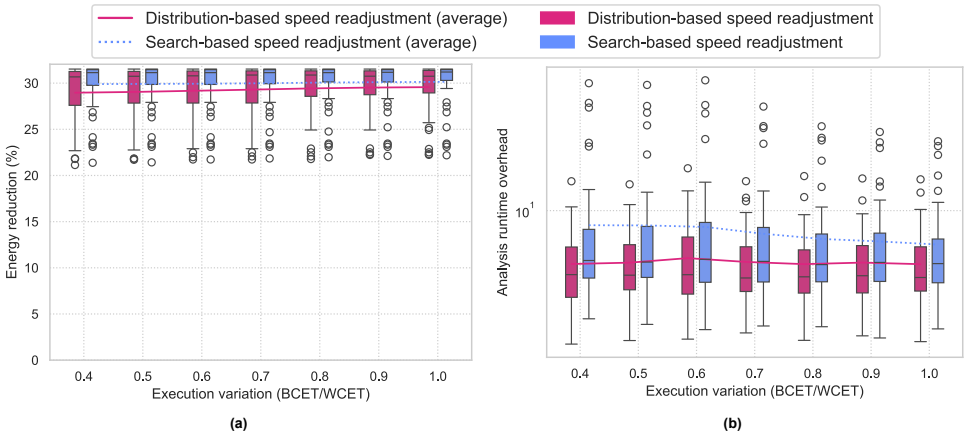


Figure 6.12: Impact of execution time variation on (a) energy reduction and (b,c,d) analysis runtime overhead with various speed readjustment methods.

($U = 40\%$), and the execution time variation for the task set was altered from 0% to 60%. For example, when the execution time variation was set to 60%, C_i^{min} for each task was set to $0.4 \times C_i^{max}$.

As observed in figs. 6.12(a) and 6.12(b), reducing the best-case to worst-case execution time ratio results in slightly higher energy reduction and marginally lower analysis runtime overhead for both speed readjustment approaches. Increasing the value of the best-case to worst-case execution time ratio decreases the execution time variation, which further reduces the number of possible execution scenarios attributed to timing uncertainty, resulting in a lower number of execution scenarios explored in the schedule abstraction graph. This reduction in execution scenarios leads to fewer causal connections and blocking scenarios for a job, thereby lowering exploration overhead while allowing for slightly improved energy reduction.

7

Conclusion

7.1. Summary of contributions

We introduced a novel energy-aware online global scheduling approach using a non-preemptive job-level fixed-priority (JLFP) policy on a homogeneous multi-core platform with arbitrary cores and discrete speed settings. To the best of our knowledge, this is the first solution to address this problem, employing static slack reclamation to assign energy-efficient speeds to each job while maintaining schedulability.

To address the main research question defined in section 1.3, the presented approach uses a schedule abstraction graph (SAG), a reachability-based response time analysis tool, to perform schedulability analysis with the selected speeds. The core idea of our energy-aware scheduling method is to explore various execution scenarios with SAG by initially assigning all jobs in the job set at the lowest speed and then readjusting the speeds if a deadline violation is detected. To optimize this process, we identify connections between jobs based on execution scenarios, which helps narrow the search space for speed adjustments.

This method ensures that the energy-aware speed assignments maintain the schedulability of the job set under the chosen JLFP scheduling policy on a multi-core platform.

7.2. Conclusions

Coming back to the research questions **SQ 1** to **SQ 3** defined in section 4.2, in this dissertation, we showed that

SQ 1 We present an ultimate SAG to explore all possible execution scenarios for a given set of discrete speeds by adjusting the execution time intervals for each job to reflect the effect of all feasible speeds.

SQ 2 For any scheduled job $J \in \mathcal{J}$, we explore all possible execution scenarios in ultimate SAG and use rules 2 and 3 to generate a set of previously scheduled jobs that impact the response time of J based on the start time and finish time of jobs in ultimate SAG.

SQ 3 After exploring all possible execution scenarios in ultimate SAG, we generate causal links for job $J \in \mathcal{J}$ which are limited to the set of previously scheduled jobs that impact the response time of J and perform speed readjustment with search-based or distribution-based approach.

We examined the impact of different workloads and analysis-specific parameters on energy reduction and runtime overhead. Our analysis revealed that the task set utilization significantly impacts energy reduction, as higher utilization leads to reduced static slack. Additionally, the skewness in task utilization distribution affects energy reduction performance due to the limited potential for energy savings. Other workload and analysis-specific parameters had minimal impact on energy reduction. The analysis-specific parameters are also interconnected which can be leveraged to achieve a trade-off between energy reduction and runtime overhead.

For increasing the value of utilization with $m = 4$, $n = 3$ and speed range $\{0.74, 0.80, 0.87, 0.94, 1.00\}$ (as discussed in section 6.2), our solution, on average, achieved energy reduction of 26.3% using the search-based speed readjustment approach and 25.85% using distribution-based speed readjustment heuristic, all while maintaining the schedulability. Although the search-based approach offered a slightly greater reduction in energy consumption, it came with a significantly higher runtime overhead. Specifically, the average runtime overhead for the search-based approach was 82.6 times higher than that of the schedulability analysis when running all jobs at the highest speed, compared to an overhead of 7.7 with the distribution-based approach. The lowest average runtime overhead of 3.8 was achieved with the timeout speed readjustment (referred to as All-connected-high speed readjustment) with an energy reduction of 22.5%.

In conclusion, energy-aware speed assignment using a schedule-abstraction graph has demonstrated substantial energy reduction potential with static slack reclamation.

7.3. Future work

The power and energy model used in our presented work does not consider idle energy consumption, i.e., the energy consumption of the core when no jobs are executing. While considering the idle energy consumption, our work can be extended with an integrated DVFS and DPM solution.

One of the approaches can include speed assignment based on a trade-off between energy reduction with DVFS and DPM. When the slack time is short and jobs cannot be delayed, the system could reduce speed via DVFS to

save energy while still meeting deadlines. Conversely, when there is ample slack and the required speed to utilize this slack is below the critical speed, the system could reduce speed to the critical speed and then use DPM to put the processor into sleep mode during the remaining idle time.

In scenarios where DPM offers various sleep modes with increasing energy savings, a trade-off between DVFS and DPM can be achieved by selecting a sleep mode with higher energy savings instead of further reducing speed. However, DPM involves a noticeable switching time, which should be factored in when choosing the sleep mode.

The energy overhead of the DVFS speed transition can also be considered for the energy assignment to avoid executing jobs at higher speeds when the transition overhead outweighs the energy savings.

Our proposed speed model currently assumes core-level DVFS, where the core's speed can be adjusted to match the job's speed assignment. While this offers high granularity in the speed changes, core-level DVFS increases the implementation complexity of the hardware platform. Therefore, alternative DVFS models, such as assigning a fixed speed to cores or tasks, could be explored.

Since our solution performs static slack reclamation based on the worst-case execution time (WCET), any potential energy savings from differences between WCET and actual execution time remain unclaimed. To address this, a dynamic slack reclamation approach could be developed, using our solution as the default speed assignment.

Finally, our solution is designed for homogeneous multi-core platforms with m cores. It can be extended to heterogeneous platforms with q clusters of identical cores, where tasks within a cluster are scheduled using a global JLFP policy. This could be implemented by running q energy-aware speed assignment analyses with respective power models and speed-to-execution time functions. Further research could explore a more complex system model, where jobs migrate between clusters based on workload requirements and energy-saving trade-offs.

References

[1] M. Nasri, G. Nelissen, and B. B. Brandenburg, “Response-time analysis of limited-preemptive parallel dag tasks under global scheduling,” vol. 133, Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, Jul. 2019, isbn: 9783959771108. doi: 10.4230/LIPIcs.ECRTS.2019.21.

[2] A. Burns and J. McDermid, “Real-time, safety-critical systems: Analysis and synthesis,” English, *Software Engineering Journal*, vol. 9, no. 6, pp. 264–281, 1994, issn: 2053-910X.

[3] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. Davis, “A comprehensive survey of industry practice in real-time systems,” *Real-Time Systems*, vol. 58, Sep. 2022. doi: 10.1007/s11241-021-09376-1.

[4] S. Sheikh and M. A. Pasha, “Energy-efficient real-time scheduling on multicores,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 19, pp. 1–25, 2020. doi: 10.1145/3399413.

[5] S. Mittal, *A survey of techniques for improving energy efficiency in embedded computing systems*, 2014. arXiv: 1401.0765 [cs.AR].

[6] F. Zhang and S. T. Chanson, “Blocking-aware processor voltage scheduling for real-time tasks,” *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 2, pp. 307–335, 2004, issn: 1539-9087. doi: 10.1145/993396.993401. [Online]. Available: https://doi.org/10.1145/993396.993401.

[7] R. Jejurikar and R. Gupta, “Energy aware non-preemptive scheduling for hard real-time systems,” 2005, pp. 21–30. doi: 10.1109/ECRTS.2005.13.

[8] J. Li, L. C. Shu, J. J. Chen, and G. Li, “Energy-efficient scheduling in nonpreemptive systems with real-time constraints,” *IEEE Transactions on Systems, Man, and Cybernetics Part A:Systems and Humans*, vol. 43, pp. 332–344, 2 2013, issn: 10834427. doi: 10.1109/TSMCA.2012.2199305.

[9] G. A. Moreno and D. D. Niz, “An optimal real-time voltage and frequency scaling for uniform multiprocessors,” 2012, pp. 21–30. doi: 10.1109/RTCSA.2012.51.

[10] W. Y. Shieh and C. C. Pong, “Energy and transition-aware runtime task scheduling for multicore processors,” *Journal of Parallel and Distributed Computing*, vol. 73, pp. 1225–1238, 9 2013, issn: 07437315. doi: 10.1016/j.jpdc.2013.05.003.

[11] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo, "Energy-aware scheduling for real-time systems: A survey," *ACM Transactions on Embedded Computing Systems*, vol. 15, 1 Feb. 2016, issn: 15583465. doi: 10.1145/2808231.

[12] V. Peluso, R. G. Rizzo, A. Calimera, E. Macii, and M. Alioto, "Beyond ideal dvfs through ultra-fine grain vdd-hopping," in *VLSI-SoC: System-on-Chip in the Nanoscale Era – Design, Verification and Reliability*, T. Hollstein, J. Raik, S. Kostin, A. Tšertov, I. O'Connor, and R. Reis, Eds., Cham: Springer International Publishing, 2017, pp. 152–172, isbn: 978-3-319-67104-8.

[13] S. Z. Sheikh and M. A. Pasha, *Energy-efficient multicore scheduling for hard real-time systems: A survey*, Jan. 2019. doi: 10.1145/3291387.

[14] K. Jeffay, D. Stanat, and C. Martel, "On non-preemptive scheduling of period and sporadic tasks," in *[1991] Proceedings Twelfth Real-Time Systems Symposium*, 1991, pp. 129–139. doi: 10.1109/REAL.1991.160366.

[15] M. Nasri, G. Nelissen, and B. B. Brandenburg, "A response-time analysis for non-preemptive job sets under global scheduling," vol. 106, Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, Jun. 2018, isbn: 9783959770750. doi: 10.4230/LIPIcs.ECRTS.2018.9.

[16] A. Mok and W.-C. Poon, "Non-preemptive robustness under reduced system load," in *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, 2005, 10 pp.–209. doi: 10.1109/RTSS.2005.31.

[17] A. Bastoni, B. Brandenburg, and J. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers," Nov. 2010, pp. 14–24. doi: 10.1109/RTSS.2010.23.

[18] M. Nasri and B. B. Brandenburg, "Offline equivalence: A non-preemptive scheduling technique for resource-constrained embedded real-time systems (outstanding paper)," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 75–86. doi: 10.1109/RTAS.2017.34.

[19] B. Korte and J. Vygen, *Combinatorial Optimization: Theory and Algorithms*, 5th. Springer Publishing Company, Incorporated, 2012, isbn: 3642244874.

[20] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, 2006, pp. 101–110. doi: 10.1109/RTSS.2006.10.

[21] N. Guan, W. Yi, Q. Deng, Z. Gu, and G. Yu, "Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling," *Journal of Systems Architecture*, vol. 57, no. 5, pp. 536–546, 2011, Special Issue on Multiprocessor Real-time Scheduling, issn: 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2010.08.003. [Online].

Available: https://www.sciencedirect.com/science/article/pii/S1383762110001037.

[22] J. Lee, K. Koh, and C.-G. Lee, "Multi-speed dvs algorithms for periodic tasks with non-preemptible sections," in *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, 2007, pp. 459–468. doi: 10.1109/RTCSA.2007.50.

[23] K. Funaoka, S. Kato, and N. Yamasaki, "Energy-efficient optimal real-time scheduling on multiprocessors," in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2008, pp. 23–30. doi: 10.1109/ISORC.2008.19.

[24] S. H. Funk and A. Meka, "U-llref: An optimal scheduling algorithm for uniform multiprocessors," in *The 9th Workshop on Models and Algorithms for Planning and Scheduling Problems*, Citeseer, 2009, p. 262.

[25] M. A. E. Sayed, E. S. M. Saad, R. F. Aly, and S. M. Habashy, "Energy-efficient task partitioning for real-time scheduling on multi-core platforms," *Computers*, vol. 10, pp. 1–21, 1 Jan. 2021, issn: 2073431X. doi: 10.3390/computers10010010.

[26] D. Zhu, R. Melhem, and B. Childers, "Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, pp. 686–700, 7 Jul. 2003, issn: 1045-9219. doi: 10.1109/TPDS.2003.1214320. [Online]. Available: http://ieeexplore.ieee.org/document/1214320/.

[27] W.-Y. Shieh and B.-W. Chen, "Energy-efficient tasks scheduling algorithm for dual-core real-time systems," in *2010 International Computer Symposium (ICS2010)*, 2010, pp. 568–575. doi: 10.1109/COMPSYM.2010.5685446.

[28] Z. Guo, A. Bhuiyan, D. Liu, A. Khan, A. Saifullah, and N. Guan, "Energy-efficient real-time scheduling of dags on clustered multi-core platforms," vol. 2019-April, Institute of Electrical and Electronics Engineers Inc., Apr. 2019, pp. 156–168, isbn: 9781728106786. doi: 10.1109/RTAS.2019.00021.

[29] J. Chen, Y. He, Y. Zhang, P. Han, and C. Du, "Energy-aware scheduling for dependent tasks in heterogeneous multiprocessor systems," *Journal of Systems Architecture*, vol. 129, p. 102 598, Aug. 2022, issn: 13837621. doi: 10.1016/j.sysarc.2022.102598.

[30] H. Chniter, O. Mosbahi, M. Khalgui, M. Zhou, and Z. Li, "Improved multi-core real-time task scheduling of reconfigurable systems with energy constraints," *IEEE Access*, vol. 8, pp. 95 698–95 713, 2020, issn: 21693536. doi: 10.1109/ACCESS.2020.2990973.

[31] S. K. Baruah, "The non-preemptive scheduling of periodic tasks upon multiprocessors," *Real-Time Syst.*, vol. 32, no. 1–2, pp. 9–20, 2006, issn: 0922-6443. doi: 10.1007/s11241-006-4961-9. [Online]. Available: https://doi.org/10.1007/s11241-006-4961-9.

[32] N. Guan, W. Yi, Z. Gu, Q. Deng, and G. Yu, “New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms,” in *Proceedings of the 2008 Real-Time Systems Symposium*, ser. RTSS ’08, USA: IEEE Computer Society, 2008, pp. 137–146, isbn: 9780769534770. doi: 10.1109/RTSS.2008.17. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1109/RTSS.2008.17.

[33] S. Baruah, “Techniques for multiprocessor global schedulability analysis,” in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, IEEE, 2007, pp. 119–128.

[34] J. Lee, “Improved schedulability analysis using carry-in limitation for non-preemptive fixed-priority multiprocessor scheduling,” *IEEE Transactions on Computers*, vol. 66, no. 10, pp. 1816–1823, 2017. doi: 10.1109/TC.2017.2704083.

[35] A. Wieder and B. B. Brandenburg, “On spin locks in autosar: Blocking analysis of fifo, unordered, and priority-ordered spin locks,” in *Proceedings of the 2013 IEEE 34th Real-Time Systems Symposium*, ser. RTSS ’13, USA: IEEE Computer Society, 2013, pp. 45–56, isbn: 9781479920068. doi: 10.1109/RTSS.2013.13. [Online]. Available: https://doi.org/10.1109/RTSS.2013.13.

[36] M. Nasri and B. B. Brandenburg, “An exact and sustainable analysis of non-preemptive scheduling,” IEEE, Dec. 2017, pp. 12–23, isbn: 978-1-5386-1415-0. doi: 10.1109/RTSS.2017.00009. [Online]. Available: http://ieeexplore.ieee.org/document/8277276/.

[37] J.-J. Chen, H.-R. Hsu, and T.-W. Kuo, “Leakage-aware energy-efficient scheduling of real-time tasks in multiprocessor systems,” in *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS’06)*, 2006, pp. 408–417. doi: 10.1109/RTAS.2006.25.

[38] S. Park, J. Park, D. Shin, *et al.*, “Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, pp. 695–708, 5 2013, issn: 02780070. doi: 10.1109/TCAD.2012.2235126.

[39] T. Burd and R. Brodersen, “Energy efficient cmos microprocessor design,” in *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, vol. 1, 1995, 288–297 vol.1. doi: 10.1109/HICSS.1995.375385.

[40] A. Chandrakasan, S. Sheng, and R. Brodersen, “Low-power cmos digital design,” *IEEE Journal of Solid-State Circuits*, vol. 27, no. 4, pp. 473–484, 1992. doi: 10.1109/4.126534.

[41] D. Zhu, D. Mosse, and R. Melhem, “Power-aware scheduling for and/or graphs in real-time systems,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 15, pp. 849–864, Oct. 2004. doi: 10.1109/TPDS.2004.45.

[42] G. Aupy, A. Benoit, P. Renaud-Goud, and Y. Robert, “Energy-aware algorithms for task graph scheduling, replica placement and checkpoint strategies,” in *Handbook on Data Centers*, S. U. Khan and A. Y. Zomaya, Eds. New York, NY: Springer New York, 2015, pp. 37–80, isbn: 978-1-4939-2092-1. doi: 10.1007/978-1-4939-2092-1_2. [Online]. Available: https://doi.org/10.1007/978-1-4939-2092-1_2.

[43] D. H. P. C. C. (DHPC), *DelftBlue Supercomputer (Phase 2)*, https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2, 2024.

[44] P. Emberson, R. Stafford, and R. Davis, “Techniques for the synthesis of multiprocessor tasksets,” *WATERS’10*, Jan. 2010.

[45] R. Stafford, *Random vectors with fixed sum*, https://www.mathworks.com/matlabcentral/fileexchange/9700-random-vectors-with-fixed-sum, MATLAB Central File Exchange, 2024. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/9700-random-vectors-with-fixed-sum.