

Department of Electrical, Computer, and Software Engineering

Part IV Research

Project

Project Number: 116

How much slack is in a multi-processor schedule?


Prabhnandan Singh

Oliver Sinnen, Supervisor

15/10/2021

Declaration of Originality

This report is my own unaided work and was not copied from nor written in collaboration with any other person.

Signature: 

Name: Prabhchandran Singh

Table of Contents

1. Introduction	5
2. Literature Review	5
2.1. Review Methodology and Analysis.....	6
2.2. Calculating Slack time.....	6
2.3. Slack Reclamation	7
2.3.1. Dynamic Voltage/Frequency Scaling	7
2.3.2. Slack management algorithms.....	8
2.3.3. Procrastination scheduling and leakage power.....	8
3. Research objectives	9
4. Model and Definitions	9
4.1. Task Graph	10
4.2. DAG Schedule Model	10
4.3. Slack Time.....	10
4.4. Idle Time	11
4.5. An Example	11
4.6. Slack time and Idle time calculation algorithm	12
5. Scheduling	13
5.1. List Scheduling.....	13
5.2. Cluster Scheduling.....	13
5.3. Comparison.....	14
6. Methodology.....	14
6.1. Process Overview	14
6.2. Graph Structures.....	15
6.3. Scheduling techniques	15
6.3.1. List-scheduling parameters.....	15
6.3.2. Cluster-scheduling parameters	16
6.4. Comparison Metrics	16
7. Experimental Evaluation	17
7.1. Data Analysis and filtration.....	17
7.2. Slack and Graph Structures	18
7.3. Slack and Scheduling Algorithms	19
7.4. Slack and Efficiency.....	20
7.5. Slack and CCR	20
7.6. Idle Time	21
7.7. Slack Chunk Size.....	22
8. Discussion.....	23
9. Conclusion.....	24
10. References	Error! Bookmark not defined.

Table of Figures

Figure 1: A DAG graph	11
Figure 2: An example of a schedule	11
Figure 3: Boxplot of Slack Time % by Graph Structure	18
Figure 4: Boxplot of Slack Time % by the algorithm used	19
Figure 5: Slack Time % by efficiency	20
Figure 6: Idle Time % by efficiency	20
Figure 7: Number of schedules by CCR	20
Figure 8: Boxplot of Slack Time % by CCR	20
Figure 9: Boxplot of Idle Time % by Graph Structure	21
Figure 10: Boxplot of Idle Time % by algorithm	21
Figure 11: Boxplot of Slack Chunk % by graph structure	22
Figure 12: Boxplot of Slack Chunk % by algorithm	22

Table of Tables

Table 1: Graph structures by category	15
Table 2: Priority and Placement schemes summary	15
Table 3: Clustering, Merging and Ordering schemes summary	16
Table 4: Summary of Comparison metrics	16
Table 5: Summary of updated metrics	17
Table 6: Summary of data filtration	17

Abstract—Power needs for parallel computing has immensely increased in the past few years. Many researchers have proposed scheduling algorithms to schedule multiple tasks on multiple processors efficiently. Despite this, all the time on the processors cannot be used efficiently due to communication delays between the different tasks. These gaps in the processors are generally referred to a slack. Slack allows flexibility in the schedules because some tasks can have a delayed or slower execution without affecting the makespan of the schedule. On the contrary, some of these unused times cannot be used due to precedence constraints, which we refer to as idle time. Many slack reclamation algorithms have been introduced in the past, which use this slack to reduce the energy consumption during the task execution. However, there is little to no research done around - how much slack is present in a multi-processor schedules and what properties of the scheduling process affect the amount of slack and idle time? We investigated thousands of graphs scheduled on different numbers of processors, using several list and cluster scheduling algorithms. The results show that most of the schedules had very little slack in them. In contrast, a considerable amount of idle time is present in many schedules. Our analysis can be helpful for future researchers to target energy-saving algorithms in those situations where enough slack and idle time is found.

1. Introduction

The performance of modern processors has increased immensely in the past few years. With the increasing performance, the power consumption has also increased. This increase in power consumption raises many new problems like higher costs (electrical and management) [1]. Multiple hardware (for example, power-scalable clusters) and software (for example, slack reclaiming algorithms) techniques have been developed to reduce these problems.

One of the most common software methods for energy reductions is slack reclamation. Slack time is generally referred to as the time between tasks when the processor/s are sitting idle. In multi-processor scheduling, the tasks can be assigned to different processors for parallel computing to increase the system's performance. However, in a precedence-constrained environment, some tasks cannot start execution until some other task/s that they depend on finish their execution. Furthermore, the inter-task communication overhead delays the tasks' start times and affect the schedule's efficiency [2]. This introduces slack in the schedule. In real-time systems, slack can also be found due to the difference between the actual execution time (AET) and the estimated worst-case execution time (WCET). Many researchers have developed algorithms to reclaim this slack using multiple techniques like slowing down some tasks' execution or switching processors off to reduce energy consumption. However, not much research has been done about how much slack is present in a schedule. If most schedules do not have enough slack, unnecessary overhead may be introduced by using slack reclamation techniques. Investigating the amount of slack and the effects of different properties of schedules on the amount can assist with designing future algorithms and directing slack reclamation techniques towards conditions where they can be more efficient.

2. Literature Review

In this section, we will look at some of the existing research that has been done around the topic. We will mainly be looking at any attempts made to calculate the amount of slack in multi-processor schedules and what approaches have been discovered to use the slack in schedules for better efficiency. It will help us develop the research objectives to further contribute to the research done around slack times.

2.1. Review Methodology and Analysis

Google scholar was mainly used to find different papers, as it provides a broader database. ScienceDirect, Citeseer and the University of Auckland Library database were also used using the same search terms. The terms that were used to find the papers were: “Slack time”, “Slack Reclamation Algorithm”, “DVFS scheduling”, “Optimal multi-processor scheduling”, “calculating Slack time in a schedule”, “slack assessment in multi-processor”, “list scheduling”, “cluster scheduling” and “list scheduling vs cluster scheduling”. The papers were then selected by reading their abstracts and seeing if they related to the research topic. We also looked at how often other papers cited a paper, indicating the paper’s importance in the research topic. We also looked into the “Cited By” link and used search terms (for example, slack) in only those resulting articles, allowing us to find newer information. Some of the papers were also found on the Related articles page.

We did not restrict the range of years the literature was selected from, as the older papers provided important background information while the more recent papers explained some of the advanced techniques developed to use slack in a schedule. It was found that there was not much interest in the subject before 2000, which significantly increased around 2005.

2.2. Calculating Slack time

Many publications have claimed to use slack time in a schedule to increase the efficiency of their proposed scheduling algorithms (more details in Section 2.3). However, very few publications have defined a proper algorithm to calculate the amount of slack in a schedule. [1] proposed an energy reduction algorithm for a DAG model, which has slack calculation as one of the steps. The slack is calculated for every task using the earliest start time and the latest end time. However, the algorithm does not contemplate if there is enough slack in the schedule to reclaim. If there is an insignificant amount of slack in an extensive schedule, calculating the slack for every task could result in unnecessary overhead. [3] proposed a slack calculation theorem, which calculates slack in the schedule between any two references of time in the schedule. However, the theorem only applies to real-time systems. [4] also proposes an algorithm for slack calculation, but is also applicable to real-time systems, modelled as task models.

Furthermore, most publications have not consider the times on the schedule that cannot be reclaimed. We refer to these times as idle times, as opposed to slack time. Even though idle time in a schedule cannot be reclaimed like slack time, it can still be beneficial for energy-saving, as the processors can be turned off completely during these times.

2.3. Slack Reclamation

Most of the current research is done around reclaiming the slack using various techniques. Some of the techniques are discussed in this section.

2.3.1. Dynamic Voltage/Frequency Scaling

When task scheduling is concerned, the primary objective is to execute a program faster. However, reducing energy consumption is becoming a significant objective too. One very well-known and well-examined existing technique to reduce energy consumption is dynamic voltage/frequency scaling (DVFS). DVFS is an efficient power management system that enables processors to dynamically adjust voltage supply levels to reduce power consumption at the expense of clock frequencies [5]. The slack present in a schedule can be reclaimed to slow down the execution of some tasks using DVFS, resulting in lower power consumption. Multiple algorithms have been proposed in the reviewed literature that uses the DVFS technique. However, using this technique implies a trade-off between the quality of schedules and energy consumption. [5] proposed an energy-conscious algorithm that explicitly balances the two performance metrics (makespan and energy consumption). [1] proposed an energy reduction algorithm that, using DVFS, selects the lower voltage and frequency uniformly along with the critical path in a task graph. [6] proposes two power-aware scheduling algorithms that also use slack to reduce the execution speed of future tasks, but in different environments – with and without precedence constraints. A study is also performed to analyse the effect of discrete voltage/speed levels on energy savings. It was concluded that the effect of voltage/speed adjustment overhead on the energy savings was relatively small.

2.3.2. *Slack management algorithms*

When tasks are scheduled statically, an estimated execution time is used. However, the estimated time in practice can be overestimated or underestimated, meaning some tasks might finish their execution earlier or later than expected during actual execution. For overestimations, extra available slack can be added to future tasks. In case of underestimation, this extra slack can reduce the possibility of missing the deadline. [7] presented novel slack allocation algorithms that use slack in the schedule to deal with such situations. In [8], the authors used the slack time to increase the efficiency of their proposed scheduling algorithm. The authors have proposed a Least slack time rate first algorithm, which overcomes some of the existing scheduling algorithm's (EDF, LRT and LST) limitations in a multi-processor environment. They have also demonstrated that their algorithm could show optimal possibility. [9] has presented a set of principles for effective slack management in EDF-based systems, which help reduce deadline miss ratio and tardiness.

2.3.3. *Procrastination scheduling and leakage power*

In addition to dynamic power consumed during task execution, [10] talks about another source of power consumption: leakage power. Leakage power is consumed as long as there is an electric current in the circuits, which include slack time and idle time in the schedules. The suggested algorithms (Dynamic Repartitioning and Dynamic Core Scaling) in [10] reduce dynamic power consumption and leakage power consumption. Dynamic Core Scaling deactivates excessive cores by exporting their assigned task to other activated cores, thus reducing leakage power consumption. The evaluation done in [10] shows that Dynamic Core Scaling saves much more power than Dynamic Repartitioning. [11] uses procrastination scheduling and a dynamic slowdown technique in conjunction to reduce the leakage power consumption. In procrastination scheduling, task execution is delayed, maximising idle time intervals, allowing the processors to sleep for longer intervals. The algorithm proposed in [11] distributes the slack between slowdown and procrastination instead of using the entire slack for just one, which is proclaimed to be more energy efficient. The scheduling technique proposed in [12] (EAPSM) also uses a similar approach optimising the overhead using task migration and frequency switch. EAPSM is integrated with Task processor affinity metric and Processor frequency affinity

metric techniques to minimise the overhead effects. Procrastination techniques were the only techniques that used both the slack time and idle time in the schedules for energy efficiency.

3. Research objectives

The previous section looked at how various task scheduling algorithms and techniques use slack for more efficient performance. Many researchers have investigated slack reclamation. However, it is possible that the suggested techniques by the researchers only work under certain situations that they have targeted. Moreover, it is crucial to analyse if there is enough slack in current schedules for these techniques to be helpful, rather than just causing excessive overhead. Additionally, we want to raise awareness about the amount of idle time found in the schedules. Therefore, it is vital to analyse how different aspects of scheduling affect the amount of slack in a schedule. All these gaps in the current research have led us to some questions, answering which are the main objectives of our research, and are listed below:

1. Which graph structures produce more slack compared to other graph structures?
2. How is the amount of slack in a schedule affected by the algorithm used to create the schedule?
3. Does the amount of slack depend on any other properties of the schedule?
4. Is it worthwhile to consider idle time in the schedules for slack reclamation?

Before answering the above questions, it is essential to define slack formally. As seen in the previous section, not much research is done around defining and calculating slack, especially in a DAG model. Furthermore, since not all the unoccupied time on a processor can be reclaimed as slack, it is vital to distinguish and introduce the concept of idle time. We believe many energy-aware scheduling techniques could benefit from using idle time. Therefore, the following sub-objectives are also introduced to the scope of our research:

1. What is slack time in a schedule?
2. What is idle time, and how is it different from slack time?
3. How to calculate the total slack time and idle time in a schedule?

4. Model and Definitions

Our research will be focused on static schedules with precedence constraints on homogeneous processors. When modelling the different tasks in a precedence-constrained application, task graphs or DAG (directed

acyclic graph) models are generally the most suitable approach. This section describes the model used in the research, formally defines slack time and idle time, and proposes an algorithm that can be used to calculate the total slack time and idle time in a DAG-based multi-processor schedule.

4.1. Task Graph

Generally, a precedence-constrained application is be modelled as a task graph or DAG, $G(V, E)$, where V is a set of n nodes $V = \{v_0, \dots, v_{n-1}\}$ and E is a set of directed edges. The nodes represent the parallel tasks. An edge $e(i, j) \in E$ from task v_i to v_j represents a precedence constraint. The weight on task v_i is denoted by w_i which represents the computation cost of that task. The weight of an edge $e(i, j)$ is denoted as $c_{i,j}$ which represents the communication cost from v_i to v_j . The communication cost between two tasks is only needed if the tasks are assigned to different processors (model taken from [13] and [5]). The publications [1], [14], [13], [5], [2] and [15] all use an application model similar to the one stated above and are used in a similar application problem.

Some other publications were also found which use task models as their application models [11], [16], [10], [12], [17], [18], [19], [6] and [20]. However, task models are generally more suitable for modelling tasks in a real-time system, as observed in the reviewed literature.

4.2. DAG Schedule Model

Tasks modelled as DAG can be scheduled on multiple processors. The schedules can also be modelled as DAGs. Schedules in this paper are modelled as a DAG (T, E, P) . T is a set of tasks that are scheduled on different processors which belong to a set P . A task i in T is denoted as t_i and has t_{i^0} and t_{i^1} as the start time and the finish time. $t_{next(i)}$ refers to the task that is scheduled right after the task t_i on the same processor. A specific processor i in P is denoted as p_i . T_{p_i} is a set of all the tasks which are scheduled on the processor p_i . The processor on which a task t_i is scheduled on is denoted as p_{t_i} . The communication cost between two tasks t_i and t_j is denoted as e_{t_i, t_j} and belongs to the set of edges, E .

4.3. Slack Time

The formal definition for slack time, as used in this paper, is given below.

Definition 4.1 (Slack time for a task). Slack time for a task is defined as the maximum amount of time the task can be delayed, provided the start times of other tasks remain unchanged.

An important assumption made is that all the tasks are assumed to be scheduled as early as possible.

S denotes all the slack times s_i on all the processors in the schedule. The slack times for a particular processor, say p_i is denoted as S_{p_i} . The slack time in S for a task t_i is denoted as s_{t_i} and has $s_{t_i^0}$ and $s_{t_i^1}$ as the start and the finish time, respectively.

Equation (1) shows the calculation of slack time for a task.

$$s_{t_i} = \min \left(t_{next(i)^0} - t_{i^1}, \min_{t_j \in T | e_{t_i, t_j} \in E} (t_{j^0} - t_{i^1} - e_{t_i, t_j}) \right),$$

where $p_{t_i} = p_{t_{i+1}}, p_{t_i} \neq p_{t_j}$

(1)

According to equation (1), a non-zero value for slack time denotes that the task has slack and can be delayed for that time without affecting the scheduled times for other tasks and the total makespan of the schedule.

4.4. Idle Time

Due to constraints from communication between tasks and unavailability of tasks, the total slack time in a schedule is limited. The rest of the unused time on the processors is referred to as Idle Time. Hence the definition of Idle Time can be given as below.

Definition 4.2 (Idle Time in a schedule). Idle time is all the unused time on the processors due to unavailability of tasks or precedence constraints on the start time of the available tasks.

In other words, idle time in a schedule can be calculated by subtracting the total slack time from the total unused time.

4.5. An Example

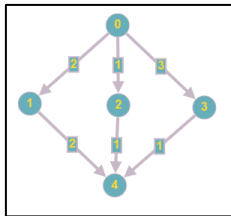


Figure 1: A DAG graph

	p_1	p_2
0	t_0	
2	t_1	t_2
4	s_{t_1}	
	t_4	

Figure 2: An example of a schedule

Figure 1 shows an example of multiple tasks modelled as a DAG, $G(V, E)$. For simplicity, all tasks are assigned a weight of 1 time unit. The weight associated with each edge is shown in the figure. An example of the

resulting schedule after scheduling the given tasks on two processors is shown in Figure 2. The schedule is modelled as our defined model (T, E, P) , where $T = \{t_0, t_1, t_2, t_3, t_4\}$, $E = \{e_{t_0, t_1}, e_{t_0, t_2}, e_{t_0, t_3}, e_{t_1, t_4}, e_{t_3, t_4}, e_{t_2, t_4}\}$ and $P = \{p_1, p_2\}$. Each task has a start and a finish time in the schedule; for example, for task t_1 , $t_{1^0} = 2$ and $t_{1^1} = 3$. Each scheduled task has a processor associated with it; for example, $p_{t_2} = p_2$. In this schedule, task t_4 cannot start any earlier due to the communication delay from the task t_2 . However, task t_1 can be delayed by one time unit without affecting the start time of t_4 , since tasks t_1 and t_4 are on the same processor. This introduces the slack time s_{t_1} in the schedule, where $s_{t_{1^0}} = 3$ and $s_{t_{1^1}} = 4$. On the other hand, task t_2 cannot be scheduled any earlier due to communication delay from t_0 , introducing idle time on p_2 between time units 0 and 2. More idle time is introduced on p_2 after the execution of t_2 due to the unavailability of tasks.

4.6. Slack time and Idle time calculation algorithm

The algorithm used to calculate slack and idle time uses the DAG model and definitions proposed in the previous subsections. The algorithm is given below.

Algorithm 1: Slack Calculation Algorithm

Input: A DAG (T, E, P) multi-processor schedule

```

1.  Let  $S$  be a list to contain all the slack times in the schedule
2.  for every processor  $p_i$  in  $P$  do
3.      Sort  $T_{p_i}$  by start times of  $t_i$ 
4.      for every task  $t_i$  in  $T_{p_i}$  do
5.          if task  $t_{next(i)}$  exists then
6.               $s_{t_{i^0}}, s_{t_{i^1}} = t_{i^1}, t_{next(i)^0}$ 
7.          else
8.               $s_{t_{i^0}}, s_{t_{i^1}} = t_{i^1}, \max_{t_j \in T}(t_{j^1})$ 
9.          end if
10.         if  $s_{t_{i^0}} \neq s_{t_{i^1}}$  then
11.             for every  $e_{t_i, t_j}$  do
12.                 if  $p_{t_i} \neq p_{t_j}$  then
13.                     if  $(t_{j^0} - e_{t_i, t_j}) < s_{t_{i^1}}$  then
14.                          $s_{t_{i^1}} = t_{j^0} - e_{t_i, t_j}$ 
15.                     end if
16.                 end if
17.             end for
18.         end if
19.         if  $s_{t_{i^0}} \neq s_{t_{i^1}}$  then
20.             Add  $s_{t_i}$  to  $S_{p_i}$ 
21.         end if
22.     end for
23.     Sort  $S_{p_i}$  by start times of  $s_i$ 

```

```

24. |   for every  $s_i$  in  $S_{p_i}$  do
25. |       if  $s_{i^1} > s_{(i+1)^1}$  then
26. |            $s_{i^1} = s_{(i+1)^1}$ 
27. |           Remove  $s_{i+1}$  from  $S_{p_i}$ 
28. |       end if
29. |   end for
30. |   Add  $S_{p_i}$  to  $S$ 
31. end for
32. for every  $s_i$  in  $S$  do
33. |   Total Slack time +=  $s_{i^1} - s_{i^0}$ 
34. end for
35. Total Idle time = Total Processor time – Total Slack time – Total Processing Time

```

5. Scheduling

When scheduling a given DAG on multiple processors, list-scheduling and cluster-scheduling are considered as the dominant approaches. These approaches are summarised in this section.

5.1. List Scheduling

List-scheduling is generally accepted as an attractive approach due to its low complexity yet good results. The different algorithms for list-scheduling usually follow a simple two-step process. First, the tasks are each assigned a priority. The priority of the tasks can be computed statically or dynamically, according to the selected priority scheme. In the second step, the tasks are placed onto the different processors, according to the priorities of the task and the placement schemes [21] [22]. Different priority and placement schemes are often used to target a specific scheduling problem. Some examples of list-scheduling algorithms include Highest Level First with Estimated Time (HLFET), Modified Critical Path (MCP), Earliest Time First (ETF) and Dynamic Level Scheduling (DLS), explained in [23].

5.2. Cluster Scheduling

Cluster-scheduling algorithms are rather complex than list-scheduling algorithms. Cluster-scheduling targets task scheduling on an unbounded number of processors and can be also be used on a limited number of processors, using some existing mapping techniques. There are three main steps involved in cluster scheduling. First, the tasks with high inter-communication are clustered together to be run on the same processor, reducing communication delays. Then, the clusters are merged to fit them on the given number of processors. Lastly, the tasks in each cluster are ordered, giving us the final schedule [24] [25]. There are also multiple schemes available for clustering, merging, and ordering the tasks during the steps. Examples of cluster scheduling

algorithms include Edge-Zeroing (EZ), Linear Clustering (LC), Dominant Sequence Clustering (DSC) and Dynamic Critical Path (DCP), explained in [26].

5.3. Comparison

Multiple studies have analyzed and compared the performance of list-scheduling ([22] [23] [21]) and cluster-scheduling ([27] [28]) algorithms separately. A thorough inter-comparison is also performed in [25]. List-schedulers were found to be better than cluster-schedulers in the majority of the cases when scheduling on a limited number of processors. Whereas cluster-schedulers were found to perform much better when scheduling to an unlimited number of processors. However, an open question is how much slack is found in the schedules generated by each type of scheduling algorithm, which is one of our research objectives.

6. Methodology

This section briefly explains the process followed to analyse the amount of slack in multi-processor schedules. The different graph structures and scheduling algorithms are also mentioned, and their selection is justified.

6.1. Process Overview

A scheduling library with multiple graphs of different graph structures was provided to us by our project supervisor. We decided to use all the graphs of all the graph structures to have a wide range of data for performing the analysis. Multiple scheduling algorithms were also available in the given library, and therefore through thorough analysis, we selected the most substantial ones for our project. Algorithm 1 was implemented and incorporated into the scheduling library. A script was implemented, which was responsible for scheduling all the graphs on a range of processors using all the selected scheduling algorithms. The slack times and the idle times were calculated for every graph structure while being scheduled. The results with other relevant information were programmed to save on a CSV file automatically. A Python workbook was then used to analyse the results in the CSV file. A comprehensive process was followed to investigate how the amount of slack is affected by the different scheduling attributes.

6.2. Graph Structures

A total of 5170 graphs were included in the experiment, with a mixture of random structures, regular structures, and application structures. Table 1 lists the categorised graph structures. More information about the graphs can be found in [25].

Table 1: Graph structures by category

Category	Graph Structure
Regular structures	Fork, Fork-join, Join, Pipeline,
Random structures	Random (Erdos-Rényi), Random (layer-by-layer)
Application structures	Cholesky, Cybershake, Epigenomics, FFT, Gauss, Independent, Inspiral, Montage, Sipht, Stencil, fpppp, Robot, Sparse

The number of nodes majorly range from 4 to 250 and 500 to 750, with some graphs having up to 2079 nodes. The graphs have a Communication-to-Computation ratio (CCR) of approximately 0.1, 1, 2, 5 and 10. The CCR helps analyse the effect of communication delays when scheduling on multiple processors.

6.3. Scheduling techniques

The numbers of processors on which the graphs are scheduled are 2, 4, 8 and 20. The range of processors was selected mainly because most of the graphs had a relatively low number of nodes. List-scheduling and cluster-scheduling algorithms are used to schedule all the graphs on multiple processors. The different parameters selected for each technique are summarised in the following subsections.

6.3.1. List-scheduling parameters

List-scheduling algorithms can primarily be split into two steps – assigning priority to tasks and placing tasks on the processors. The priority and placement schemes used for our experiment are summarised in Table 2.

Table 2: Priority and Placement schemes summary

	Scheme	Description
Priority schemes	Blevel	Prioritises tasks by decreasing bottom-level
	ETF – Earliest Time First	Prioritises tasks with the smallest earliest start time (EST)
Placement schemes	Norm	Place tasks to minimise task's EST
	Weighted-children EST	Place task to minimise the sum of the task's EST and its critical child's EST

6.3.2. Cluster-scheduling parameters

Cluster-scheduling can generally be decomposed into three steps – clustering the task, merging the clusters, and ordering the tasks in the clusters. The parameters used for cluster-scheduling in our experiment are summarised in Table 3.

Table 3: Clustering, Merging and Ordering schemes summary

	Scheme	Description
Clustering schemes	DCP – Dynamic Critical Path	Uses critical path (sum of the task’s EST and the critical child’s EST) to determine the clusters for every task
	DSC – Dominant Sequence Clustering	Uses priorities (sum of top-level and bottom-level) to determine the clusters for every task [28]
Merging schemes	GLB – Guided-Load-Balancing	Merge clusters using load balancing while also considering the starting times of the tasks. [29]
	List – Merge by list scheduling	Uses priority order like list-scheduling and schedules entire cluster to the processor with minimum schedule length when merged with other clusters pre-assigned to that processor [30]
Ordering schemes	Blevel	Order tasks by decreasing bottom-level
	EST	Order tasks with smallest earliest start time

6.4. Comparison Metrics

It is unknown from former research what properties of a schedule the amount of slack depends on. Therefore, we have used a wide range of comparison metrics to find correlations between the amount of slack and the other properties. Table 4 summarises the comparison metrics considered in our research.

Table 4: Summary of Comparison metrics

Metric	Definition
Slack %	Percentage of Total Slack time / Total Processor time
Idle %	Percentage of Total Idle time / Total Processor time
Avg Slack Chunk	Sum of all slack times/ total occurrences of slack
Slack Chunk %	Percentage of Avg Slack Chunk/Total Slack Time
Speedup	Makespan of parallel schedule/Total sequential time
CCR	The ratio of the sum of weights of edges to the sum of weights of nodes
Algorithm Type	The algorithm based on list-scheduling or cluster-scheduling
Algorithm	Algorithm specific schemes used
Graph Type	Type of the graph structure
Num of Processors	Number of processors the graphs were scheduled on

Since the makespan of a schedule varies for the different schedules, we decided to use percentage proportions of slack times, idle times, and slack chunk size rather than the actual amount to consistently compare the different schedules. Some of the schedules had no tasks scheduled on some of the processors. Therefore, to

prevent these cases from skewing the analysis, we removed those processors from the total processors and introduced additional updated metrics, summarised in Table 5.

Table 5: Summary of updated metrics

Metric	Definition
Procs Used	Number of processors with at least one task scheduled on them
Slack Time %	Percentage of Total Slack time / Total Processor time of used processors
Idle Time %	Percentage of Total Idle time / Total Processor time of used processors
Efficiency	Speedup/Procs Used

7. Experimental Evaluation

This section summarises the key findings of the analysis done on the results. Due to space limitations, all the analysis details could not be discussed. However, they will be made available in the python notebook provided in the research compendium.

7.1. Data Analysis and filtration

The main aim of multiprocessing is to increase the speed of execution compared to sequential execution. However, there are some cases when using multiple processors does not provide a speedup. We found such cases in our data as well. We assumed that schedules that do not provide a speedup would not be used in practice. Therefore, these data entries were discarded to prevent them from distorting the results of the analysis. A summary of the original data, the discarded data and the filtered data are given in Table 6.

Table 6: Summary of data filtration

Graph Type	Original count	Discarded count	Filtered count
Cholesky	720	0	720
CyberShake	720	0	720
Epigenomics	720	7	713
FFT	720	0	720
Fork	31776	1456	30320
Fork-join	22128	2718	19410
Gauss	576	17	559
Independent	6048	0	6048
Inspiral	720	0	720
Join	33888	1636	32252
Montage	720	0	720
Pipeline	18240	5906	12334
Random	109968	14344	95624
Random (layer-by-layer)	719	0	719
Sipht	720	12	708
Stencil	19344	4061	15283
fpppp	144	0	144

Robot	144	0	144
Sparse	144	0	144
Total	248159	30157	218002

Since the number of samples for each of the graphs are different, the credibility of the conclusions drawn for each of the graph structures will depend on the number of samples. For example, since the Join graphs count is high, the conclusions drawn for our data set are highly likely to apply to all Join graph structure examples.

7.2. Slack and Graph Structures

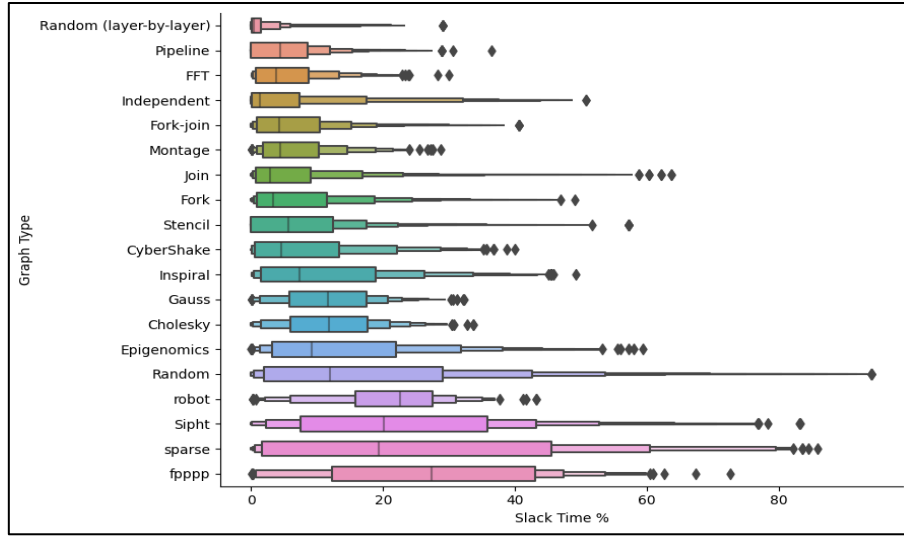


Figure 3: Boxplot of Slack Time % by Graph Structure

Figure 3 shows a boxplot of the Slack time % of the different graph structures. For the majority of the graph structures (Random (layer-by-layer) to Epigenomics), the amount of the slack in almost 75% of the schedules created was under 20%. For Random graph structures, about 40% of the schedules had a slack amount of over 20%. However, the amount of slack was still under 30% for about 75% of the schedules. For robot graph structure, 50% of the schedules had about 18% to 30% slack. The amount of slack found in Sipht and Sparse graph structure schedules varied from low slack to high slack, where about 50% of the schedules had less than 20% slack. However, for Sipht graph structures, about 25% of the schedules had a slack amount between 20% and 35%, and for Sparse graph structures, about 25% of the schedules had a slack amount between 20% and 45%. For fpppp graph structures, about 25% of the schedules had a slack amount between 15% and 25%, and the other 25% had about 25% to 43% of slack. An important observation is that more slack was found in application graph structures than the regular and random ones. However, considering the number of data

samples for each graph structure, the conclusions drawn for Fork, Fork-join, Join, Pipeline, Random and Stencil graph structures are expected to apply to all graphs outside our dataset. In contrast, the conclusions are a mere indication of the graphs outside our dataset for other graph structures.

7.3. Slack and Scheduling Algorithms

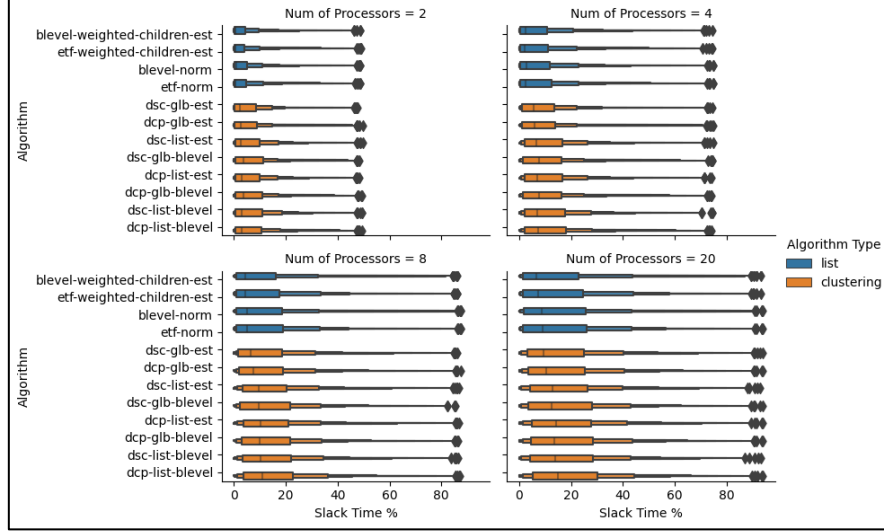


Figure 4: Boxplot of Slack Time % by the algorithm used

Figure 4 shows boxplots for Slack Time % for each algorithm used, categorised by the number of processors the graphs were initially set to be scheduled on. The amount of slack on a schedule increased with the increasing number of processors for all the different algorithms. Such behaviour is expected because the tasks are expected to be more closely scheduled with fewer processors, therefore reducing most communication delays. However, with the increasing number of processors, different tasks dependent on each other are more likely to be scheduled on different processors, causing more communication delays in the schedule. For all the different processor environments, it was found that cluster-scheduling algorithms produced a little more slack than list-scheduling algorithms. Cluster-scheduling algorithms usually target scheduling tasks on an unlimited number of processors. Therefore, the schedules created by cluster-scheduling algorithms are expected to work less efficiently on a limited number of processors when compared to list-scheduling algorithms. Therefore, it becomes imperative to compare the amount of slack with the efficiency of the schedule, discussed in the next section.

7.4. Slack and Efficiency

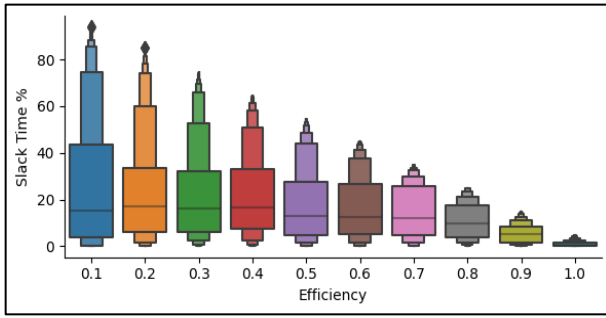


Figure 5: Slack Time % by efficiency

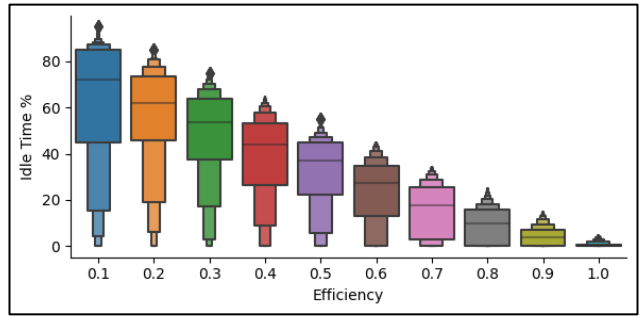


Figure 6: Idle Time % by efficiency

Figure 5 and Figure 6 show the boxplots of slack time % and idle time % by the efficiency of the schedules. Figure 5 revealed that the maximum amount of slack in a schedule was limited by the efficiency of the schedule; as the efficiency increased, the maximum amount of slack decreased. The average amount of slack in most of the schedules did not significantly get affected by the efficiency. However, from Figure 6, the amount of idle time was significantly high at lower efficiencies. Almost 50% of the schedule's time was calculated to be idle time at lower efficiency. Therefore, it can be concluded that the efficiency of a schedule only affects the maximum amount of slack in a schedule. At the same time, the idle time in a schedule is significantly affected by efficiency.

7.5. Slack and CCR

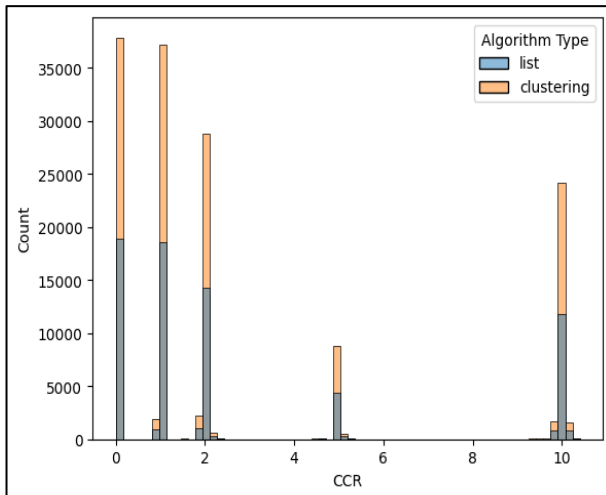


Figure 7: Number of schedules by CCR

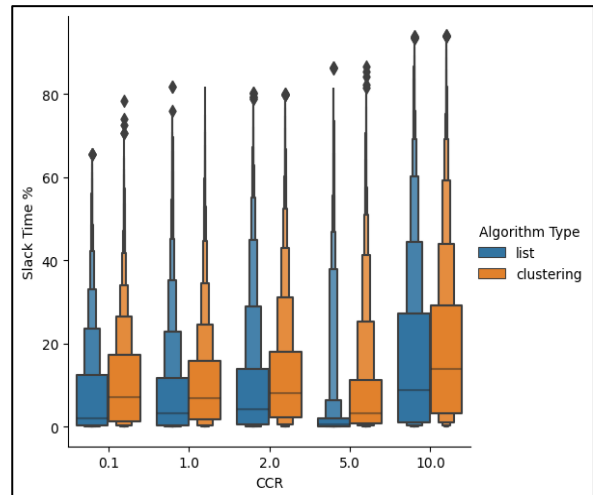


Figure 8: Boxplot of Slack Time % by CCR

Figure 7 shows the number of schedules in the dataset by CCR. The majority of the schedules have a CCR value of 0.1, 1, 2, 5 or 10. Therefore, only these values will be considered for analysing the effect of CCR on the amount of slack in a schedule. Figure 8 shows the Slack Time % for these CCR values. Since the CCR

affects how the graphs are scheduled, the boxplots are categorised by the algorithm used. There is no apparent trend for graphs with different CCR values. Graphs with a CCR of 5 have significantly low slack compared to other graphs, but this could be due to the frequency of schedules at this CCR being relatively low. Graphs with a CCR of 10 had slightly more slack than all the other graphs. Since the communication costs are much higher at a CCR of 10, the interdependent tasks being scheduled on different processors are expected to be scheduled much later than other graphs with lower CCRs. Due to this, the start times of other tasks are also expected to be delayed, producing more slack. Through further analysis, graphs with a CCR of 10 were also the least efficient of all. The scheduling algorithms do not seem to directly influence the impact of CCR on the amount of slack in the schedules.

7.6. Idle Time

Since some energy-aware algorithms consider idle time, we decided to analyse idle time in different graph structures. Figure 9 and Figure 10 show boxplots for Idle time % in the schedules by graph structures and algorithms.

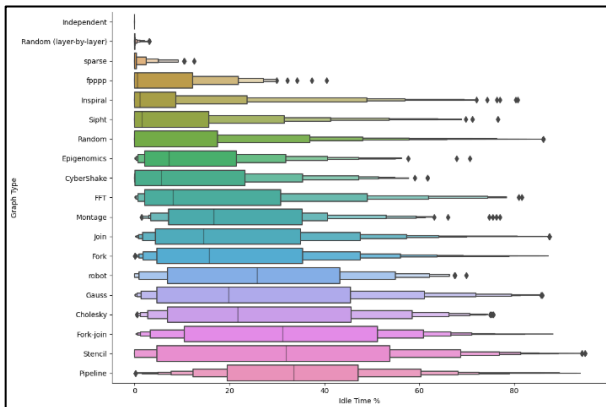


Figure 9: Boxplot of Idle Time % by Graph Structure

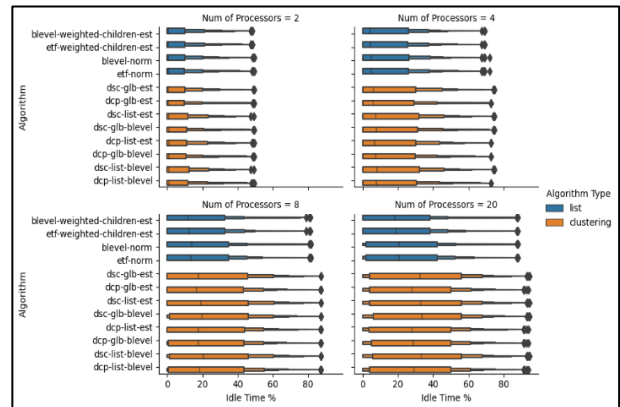


Figure 10: Boxplot of Idle Time % by algorithm

Figure 9 shows that majority of the schedules have a significantly high amount of idle time. Although, more idle time is found in regular graph structures' schedules than random and application structures. Almost 50% to 75% of the schedules of regular structures had idle time % up to 40%. Since the count of regular structures in the dataset is significantly high, it can be concluded that most of the regular graph structures produce up to 40% of idle time in the schedules. Figure 10 shows that the amount of idle time increased as more processors

were used. The amount of idle time from using different algorithms were relatively consistent within each processor environment, with cluster-schedulers producing a little more idle time than list-schedulers.

7.7. Slack Chunk Size

The previous subsections focused on the total amount of slack and idle times in the schedules. However, from the previous research, it is known that there is an extra overhead when reclaiming slack, caused by voltage/speed adjustments or switching processors on/off. Therefore, this subsection will look into the average chunk sizes for the slack times. Figure 11 and Figure 12 show boxplots for the Slack Chunk % for each graph structure and algorithm, respectively.

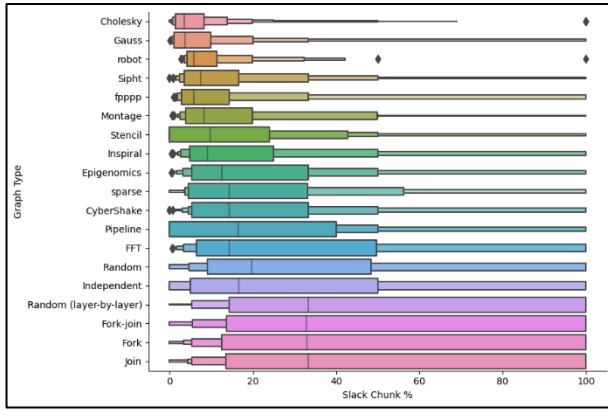


Figure 11: Boxplot of Slack Chunk % by graph structure

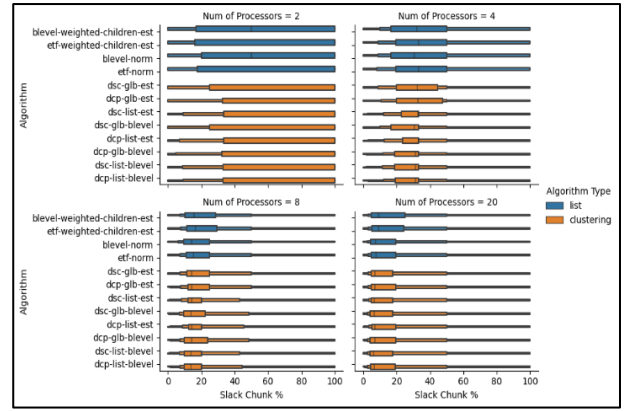


Figure 12: Boxplot of Slack Chunk % by algorithm

Figure 11 shows that about 50% of the schedules had slack chunk % under 20% for a majority of the graph structures. This means that for these graph structures, the slack is highly distributed over the schedules. However, for Random(layer-by-layer), Fork-join, Fork and Join graph structures, over 50% of the schedules had slack chunk % above 35%, going up to 100%. Meaning in these schedules, the slack is less distributed and more collective, and even in some cases, the total slack is just one chunk. The slack chunk % found in these schedules is high, possibly because there is significantly less slack in these schedules, as seen in Figure 2; therefore, there are fewer chunks of slack times than other graph structures. Figure 12 shows that bigger chunks of slack are only found when scheduling on two processors. As seen before, the schedules are more efficient on two processors, implying that the tasks are more closely scheduled, reducing slack distribution.

8. Discussion

This section summarises the different findings from Section 7, determining their significance and relating them to the previous research.

For the majority of the analysed graph structures, the amount of slack is quite low. For some application graphs, a relatively high amount of slack was found in the schedules. However, since the number of samples for these schedules is moderately low, the conclusions cannot be drawn confidently. Whereas the inverse was found when considering the amount of idle time in the schedules as a significant amount of idle time was found in most of the schedules. Regular graph structure schedules were found to have more idle time compared to application structures. Since the count of regular structures was considerably high, the conclusion drawn is more likely to apply to other graphs of such structures. The efficiency of a schedule was found to only affect the maximum amount of slack time in a schedule, decreasing with increasing efficiency. However, the efficiency of a schedule seems to affect the idle time in the schedule significantly. A relatively substantial amount of idle time is found at lower efficiencies. Cluster-schedulers were less efficient than list-schedulers; hence, more slack and idle times were found in schedules created using cluster-scheduling. The CCR of the graphs did not have a significant effect on the amount of slack in the schedules; however, it is expected that more slack is found in schedules of graphs with higher CCR.

It can be established that using slack reclamation on relatively highly efficient schedules is expected to cause unnecessary overhead, as very little slack and idle time was found in these cases. The slack reclamation techniques should be targeted more at scheduling problems where high efficiency cannot be reached. Since the makespan cannot be reduced in these cases, high energy savings could be made as more slack and idle time is expected to be found. Additionally, a significant amount of idle time was found in the schedules, even at higher efficiencies, which many slack reclamation techniques do not use. Utilising idle times in the schedules can prove to be highly beneficial when it comes to saving energy. The processors can be turned off during idle times with little overhead, avoiding leakage power (Section 2.3.3).

9. Conclusion

This study aimed to analyse the amount of slack in different schedules created from different graph structures using different scheduling algorithms. Other comparison metrics were also used to determine which aspects of a schedule affect the amount of slack. Before that, slack and idle time was formally defined, and a clear distinction was made between the two. An algorithm is also proposed to calculate the total slack time and idle time in a schedule. The slack and idle time were calculated using the definitions and algorithm and analysed thoroughly in an extensive experimental study. We believe no such research has been performed in the past. Most schedules did not have much slack, especially for regular structure graphs, mainly because modern algorithms can efficiently schedule these graphs. However, a significant amount of idle time was found in many schedules. Further analysing showed that even at medium to high efficiencies, a considerable amount of idle time was still present. Cluster-scheduling algorithms produced more slack and idle times, which was expected, as they are less efficient than list-scheduling. It was also found that higher proportions of slack and idle time are found with the increasing number of processors. To avoid unnecessary overheads, it is suggested that energy-aware techniques are not used on schedules that are known to be highly efficient. These techniques will be most effective when used on more complex, inefficient scheduling problems where more idle and slack times are found. Other researchers are also encouraged to contemplate idle times too when seeking to increase energy savings.

Acknowledgements

I would like to sincerely thank Dr Oliver Sinnen for guiding us through the research project and sharing his expert knowledge. I also appreciate him and other previous Part IV students for providing us with the resources to achieve our research objectives.

I would also like to thank my project partner, Harrison Warahi, for contributing to the research process and sharing his thoughts and ideas.

10. References

- [1] H. Kimura, M. Sato, Y. Hotta, T. Boku and D. Takahashi, "Emprical study on reducing energy of parallel programs using slack reclamation by dvfs in a power-scalable high performance cluster," in *2006 IEEE international conference on cluster computing*, 2006.
- [2] Q. Tang, L.-H. Zhu, L. Zhou, J. Xiong and J.-B. Wei, "Scheduling directed acyclic graphs with optimal duplication strategy on homogeneous multiprocessor systems," *Journal of Parallel and Distributed Computing*, vol. 138, pp. 115-127, 2020.
- [3] J. M. Urriza, F. E. Paez, R. Cayssials, J. D. Orozco and L. S. Schorb, "Low cost slack stealing method for RM/DM," *International Review on Computers and Software*, vol. 5, p. 660–667, 2010.
- [4] R. I. Davis, K. W. Tindell and A. Burns, "Scheduling slack time in fixed priority pre-emptive systems," in *1993 Proceedings Real-Time Systems Symposium*, 1993.
- [5] Y. C. Lee and A. Y. Zomaya, "On Effective Slack Reclamation in Task Scheduling for Energy Reduction.," *JIPS*, vol. 5, p. 175–186, 2009.
- [6] D. Zhu, R. Melhem and B. R. 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, 2003.
- [7] J. Kang and S. Ranka, "Dynamic slack allocation algorithms for energy minimization on parallel machines," *Journal of Parallel and Distributed Computing*, vol. 70, p. 417–430, 2010.
- [8] M. Hwang, D. Choi and P. Kim, "Least slack time rate first: New scheduling algorithm for multi-processor environment," in *2010 International Conference on Complex, Intelligent and Software Intensive Systems*, 2010.
- [9] C. Lin and S. A. Brandt, "Improving soft real-time performance through better slack reclaiming," in *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, 2005.

- [10] E. Seo, J. Jeong, S. Park and J. Lee, "Energy efficient scheduling of real-time tasks on multicore processors," *IEEE transactions on parallel and distributed systems*, vol. 19, p. 1540–1552, 2008.
- [11] R. Jejurikar and R. Gupta, "Dynamic slack reclamation with procrastination scheduling in real-time embedded systems," in *Proceedings. 42nd Design Automation Conference, 2005.*, 2005.
- [12] P. Ramesh and U. Ramachandraiah, "Energy aware proportionate slack management scheduling for multiprocessor systems," *Procedia computer science*, vol. 133, p. 855–863, 2018.
- [13] Y. Hu, C. Liu, K. Li, X. Chen and K. Li, "Slack allocation algorithm for energy minimization in cluster systems," *Future Generation Computer Systems*, vol. 74, p. 119–131, 2017.
- [14] Z. Shi, E. Jeannot and J. J. Dongarra, "Robust task scheduling in non-deterministic heterogeneous computing systems," in *2006 IEEE International Conference on Cluster Computing*, 2006.
- [15] M. Guzek, J. E. Pecero, B. Dorronsoro and P. Bouvry, "Multi-objective evolutionary algorithms for energy-aware scheduling on distributed computing systems," *Applied Soft Computing*, vol. 24, p. 432–446, 2014.
- [16] S. Midonnet, D. Masson and R. Lassalle, "Slack-time computation for temporal robustness in embedded systems," *IEEE embedded systems letters*, vol. 2, p. 119–122, 2010.
- [17] V. Kannaian and V. Palanisamy, "Energy-efficient scheduling for real-time tasks using dynamic slack reclamation," *Turkish Journal of Electrical Engineering & Computer Sciences*, vol. 27, p. 2746–2754, 2019.
- [18] V. Nelis and J. Goossens, "MORA: An Energy-Aware Slack Reclamation Scheme for Scheduling Sporadic Real-Time Tasks upon Multiprocessor Platforms," in *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2009.
- [19] T. A. AlEnawy and H. Aydin, "Energy-constrained scheduling for weakly-hard real-time systems," in *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, 2005.

- [20] H. Baek, D. Lim and J. Lee, "Proof and Evaluation of Improved Slack Reclamation for Response Time Analysis of Real-Time Multiprocessor Systems," *IEICE TRANSACTIONS on Information and Systems*, vol. 101, p. 2136–2140, 2018.
- [21] G. Q. Liu, K. L. Poh and M. Xie, "Iterative list scheduling for heterogeneous computing," *Journal of Parallel and Distributed Computing*, vol. 65, pp. 654–665, 2005.
- [22] K. D. Cooper, P. J. Schielke and D. Subramanian, "An experimental evaluation of list scheduling," *TR98*, vol. 326, 1998.
- [23] T. Hagraš and J. Janeček, "Static vs. dynamic list-scheduling performance comparison," *Acta Polytechnica*, vol. 43, 2003.
- [24] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys (CSUR)*, vol. 31, p. 406–471, 1999.
- [25] H. Wang and O. Sinnen, "List-Scheduling versus Cluster-Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, pp. 1736–1749, 2018.
- [26] Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," *Journal of Parallel and Distributed Computing*, vol. 59, p. 381–422, 1999.
- [27] A. Gerasoulis and T. Yang, "A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 16, p. 276–291, 1992.
- [28] A. Al-Rahayfeh, S. Atiewi, A. Abuhussein and M. Almiani, "Novel approach to task scheduling and load balancing using the dominant sequence clustering and mean shift clustering algorithms," *Future Internet*, vol. 11, p. 109, 2019.
- [29] A. Radulescu and A. J. C. Van Gemund, "GLB: A low-cost scheduling algorithm for distributed-memory architectures," in *Proceedings. Fifth International Conference on High Performance Computing (Cat. No. 98EX238)*, 1998.
- [30] V. Sarkar, "Partitioning and scheduling parallel programs for execution on multiprocessors," 1987.