A Social-Aware Federated Real-Time Scheduling Algorithm for Unrelated Multiprocessor Platforms

A Novel Heuristic for Processor Assignments in Real-Time Systems on Unrelated Heterogeneous Platforms

Master's thesis in Computer science and engineering

David Wilkins
Oskar Hammargren

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

A Social-Aware Federated Real-Time Scheduling Algorithm for Unrelated Multiprocessor Platforms

A Novel Heuristic for Processor Assignments in Real-Time Systems on Unrelated Heterogeneous Platforms

David Wilkins

Oskar Hammargren

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

A Social-Aware Federated Real-Time Scheduling Algorithm for Unrelated Multiprocessor Platforms
A Novel Heuristic for Processor Assignments in Real-Time Systems on Unrelated Heterogeneous Platforms
David Wilkins
Oskar Hammargren

© David Wilkins, 2022.
© Oskar Hammargren, 2022.

Supervisor: Risat Pathan, Department of Computer Science and Engineering
Examiner: Jan Jonsson, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

A Social-Aware Federated Real-Time Scheduling Algorithm for Unrelated Multiprocessor Platforms
A Novel Heuristic for Processor Assignments in Real-Time Systems on Unrelated Heterogeneous Platforms
David Wilkins
Oskar Hammargren
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Real-time systems are commonly found in the modern world, ranging from aerospace control systems to health-care equipment. Real-time systems operate under strict timing constraints, meaning each program (i.e. task) must complete before a given deadline. Thus, a Real-time scheduling algorithm needs to schedule each task such that all deadlines are guaranteed to be met. Due to the sophistication of many modern real-time applications, the workload of real-time tasks are ever increasing. This creates a demand for multiprocessor platforms that can distribute the workload among several processors. Furthermore, many multiprocessor platforms are heterogeneous, meaning they include processors of different types that offers different capabilities to different task. This allows hardware to be specialized for different types of tasks. An example of such a platform is the ARM's big.LITTLE architecture, which combines high-performance processing unit with power-efficient processors.

However, scheduling real-time tasks on multiprocessors is a difficult problem. One approach to this problem is *federated scheduling*, which divides tasks into two categories, *light* or *heavy*. Light tasks can meet their deadline using only one processor, while heavy tasks need more than one processors to meet their deadline. Thus, federated scheduling assigns a cluster of processors to each heavy task. The light tasks are then assigned to the remaining processors. This assignment problem is an intractable problem since every possible task-to-processor assignment need to be considered in order to find the optimal solution.

The current state-of-the-art in federated scheduling on heterogeneous platforms has a limitation. Namely, each task takes its preferred processors disregarding whether these processors were critical to other tasks. We fills this gap by providing a *social-aware* processor assignment algorithm. This algorithm gives each processor to the tasks that needs it the most. Our social-aware processor assignment algorithm is empirically evaluated through simulation. The performance of our algorithm is compared with the current state-of-the-art. The simulation show that our social-aware algorithm performs better in most cases.

Keywords: real-time scheduling, resource allocation, social-awareness, federated scheduling, bin-packing, heterogeneous platforms, unrelated platforms, computer science.

Acknowledgements

We would like to thank our supervisor, Risat Pathan, who gave us the opportunity to do this thesis. Risat also provided constructive feedback and insights throughout this thesis and helped us tailor this thesis to our intended audience. Additionally, we would like to thank our examiner, Jan Jonsson, who provided valuable comments on our text and words of encouragement. Finally, we would like to thank our opponent Chengzi Huang for his constructive comments.

David Wilkins and Oskar Hammargren, Gothenburg, June 2022

Contents

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 Contributions . . . 2
- 1.2 Thesis Outlook . . . 3

2 Background 5
- 2.1 System Model . . . 5
  - 2.1.1 Sequential task model . . . 5
  - 2.1.2 Multiprocessor platform models . . . 5
  - 2.1.3 Parallel task model . . . 6
  - 2.1.4 Assignment policies . . . 7
  - 2.1.5 Run-time Scheduling Policies . . . 8
  - 2.1.6 Feasibility tests . . . 8
- 2.2 Previous Work . . . 9
  - 2.2.1 Uniprocessor scheduling . . . 9
  - 2.2.2 Partitioned scheduling . . . 10
  - 2.2.3 Makespan . . . 10
  - 2.2.4 Federated scheduling . . . 11

3 Problem Formulation 13
- 3.1 Limitations . . . 15

4 Social-Aware Algorithm 17
- 4.1 Run-time Schedulers . . . 17
  - 4.1.1 Heavy Tasks . . . 17
  - 4.1.2 Light Tasks . . . 19
- 4.2 Social-Awareness . . . 19
  - 4.2.1 Social-aware clustering . . . 23
  - 4.2.2 Supportive partitioning . . . 26
- 4.3 Social-Aware Algorithm . . . 26
  - 4.3.1 Algorithmic complexity of SAPA . . . 31

5 Simulation 33
- 5.1 Experimental Setup . . . 33

5.1.1 Generating platform 34
5.1.2 Generating DAGs 34
5.1.3 Comparisons 34
5.2 Simulation Results 35
5.2.1 General Performance 36
5.2.2 Changing the number of heavy tasks 36
5.2.3 Changing the number of processors 38
5.2.4 Changing the subtask variance 39
5.2.5 Homogeneous platforms 40

6 Discussion and Conclusion **43**
6.1 Simulation Results 43
6.2 Effects of Anomalies 44
6.3 Processor Order 44
6.4 Applicability 45
6.5 Future Work 45

Bibliography **47**

A Appendix 1 **I**
A.1 Social-Aware Partitioning I
A.2 Processor Values II
A.2.1 Original processor value II
A.2.2 Speed-based processor value III
A.2.3 Workload-based processor value III
A.2.4 Fair processor value III

B Appendix 2 **V**

C Appendix 3 **IX**

List of Figures

2.1 Example of a DAG with its corresponding worst-case execution times for each node . . . 7

3.1 Different resource assignment problems . . . 15

4.1 Diagram of the elements of the Augmented Cost Function for clustering 21

5.1 Acceptance-ratio of all algorithm variations under a platform with 66% heterogeneity and varying uniformity, plotted against goal utilization . . . 37
5.2 Acceptance-ratio on an unrelated platform with varying deadline multipliers . . . 38
5.3 Acceptance-ratio with 32 processors with varying uniformity . . . 39
5.4 Acceptance-ratio of task sets with different node variance executing on a highly unrelated platform . . . 40
5.5 Acceptance-ratio of all algorithms under a homogeneous platform . . 41

B.1 Acceptance-ratio on a heterogeneous platform of 10 processors with varying degree of uniformity (columns) and increasing deadline multiplier (rows) . . . V
B.2 Acceptance-ratio on a heterogeneous platform of 32 processors with varying degree of uniformity (columns) and increasing deadline multiplier (rows) . . . VI
B.3 Acceptance-ratio on a homogeneous platform of 10 processors with varying degree of uniformity (columns) and increasing deadline multiplier (rows) . . . VII
B.4 Acceptance-ratio on a highly unrelated platform of 10 processors with varying degree of subtask variance (columns) . . . VII

List of Tables

1 Introduction

Real-time systems are commonly found in the modern world, ranging from aerospace control systems to health-care equipment[1]. What they have in common is that the correctness of the system does not only depend on the result of the execution but also the time at which this result is available, hence the system must live up to certain timing constraints. These timing constraints are usually defined by *deadlines*, which specify the latest possible time that the programs must finish its execution for the result to be considered correct. The consequences of a missed deadline can range anywhere from a minor inconvenience to complete system failure. For instance, consider an autonomous vehicle that needs to make a left turn. It will may have a few milliseconds to perform that left turn. However, if it performs the turn after one minute it would likely crash, and the behavior of the system would not be considered correct. Such delays can occur when the system needs to perform several tasks using limited resources.

A real-time system consists of *tasks* which are independent program. These tasks compete for the same resource, namely the processor(s). Therefore, these systems require a real-time kernel (from e.g. an operating system) that employs a scheduling algorithm that schedules all the tasks such that none of them miss their deadline. This scheduling algorithm decides which tasks get to execute at what time. Furthermore, timeliness guarantees can be provided by *feasibility tests*. A feasibility test is a mathematical condition that, when true, state that all task are guaranteed to finish within their deadline.

Most modern applications require the use of multiprocessors to ensure that all tasks meet their deadlines[1]. This is because many application has a large number of tasks that need to run in parallel to ensure that all deadlines are met. In addition, it is common for applications to include *parallel tasks*. Such tasks have a high workload and therefore need to be split into several parts that can be executed in parallel. However, the use of multiprocessors introduces the additional problem of determining which tasks should run on which processor.

There are different approaches to the multiprocessor scheduling problem, one of them being *federated scheduling*. The federated scheduling technique categorizes each task as either *heavy* or *light*. Heavy tasks correspond to parallel tasks that require several processors to meet their deadline. However, light tasks can execute sequentially on a single processor. Each heavy task need to be assigned a unique set of dedicated processors to execute on, in isolation from all other tasks. We denote this problem as the *clustering* problem. On the other hand, each light task must be assigned to some processor which it shares with other light tasks. We call this the *partitioning* problem. Thus, a federated scheduling algorithm need solve the

clustering problem as well as the partitioning problem.

There is much published work considering federated scheduling on homogeneous platforms (i.e. platforms where all processors are identical). However, federated scheduling on heterogeneous platforms (i.e. platforms where not all processors are identical) has been explored to a much lesser extent. Most work that does consider heterogeneous platforms only does so for specific types of heterogeneous platforms. However, Voudouris et al. provided a federated scheduling algorithm for a generalized heterogeneous platform model called *unrelated* platforms [2]. Our work improves upon the federated scheduler in [2] by introducing a new processor assignment algorithm.

1.1 Contributions

In this thesis we consider the federated scheduling problem on heterogeneous platforms. This problem consists of two sub-problems, namely clustering and partitioning. Both of these problems can be reduced to the classical bin-packing problem, which consist of packing a set of items into a set of bins such that all items fit inside some bin. The bin-packing problem is NP-hard[3], meaning every assignment combination need to be tested to reach the optimal[1] solution. Thus, the optimal solution has an exponential time complexity which is intractable for large systems. Therefore, heuristics that provide approximate solutions are required to provide practical solutions. Our solution is a processor assignment algorithm for heterogeneous platforms that executes in polynomial time. It utilizes a novel bin-packing heuristic that posses a property we call *social-awareness*. This heuristic is applied to the clustering problem on heterogeneous platforms. Furthermore, the assignment algorithm includes a partitioning algorithm that plays a supportive role in the sense that it removes tasks that don't need to be considered for clustering.

The scheduler in [2] solves clustering by giving each processor to the first task that wants it. This is *social-unaware* in the sense that a task may unknowingly deprive another task of a critical resource. In contrast, our social-aware algorithm assigns each processor to the task that needs it the most. More importantly, this is done while also considering the processing capacities of all the other remaining processors. Furthermore, the social-aware heuristic can be applied to other applications outside the scope of federated scheduling and real-time systems. More specifically, this thesis provides the following contributions:

- Provides a formal definition of social-awareness, with respect to an assignment algorithm, that considers the demands of each task for each processor.
- Provides an assignment algorithm with social-aware clustering and supportive partitioning, that provides a heuristic solution to the federated scheduling problem.
- Illustrate the effectiveness of the social-aware assignment algorithm through simulation. The performance is compared to the current state-of-the-art [2] and measured in *acceptance ratio*, which provides the percentage of systems

[1] A solution S is optimal iff, if there exist an assignment that allows all tasks to meet their deadlines, then S will also find a valid assignment

(task sets and platforms) that are successfully assigned by the algorithm. A system is successfully assigned when all tasks are given enough processing capacity to meet their deadlines.

1.2 Thesis Outlook

The remainder of this document will begin by providing the relevant background in the research area of real-time systems. Next, a formulation of the problem is provided which details its connection to the bin-packing problem. This is followed by the theoretical contributions which consist of the definition of social-awareness as well as a description of our algorithm that adheres to this definition. Subsequently, a description of the experimental setup is provided which details how the algorithm is simulated. Following, the simulation results of our algorithm is compared to that of the state-of-the-art[2]. Finally, the theory and the simulation results are discussed.

2
Background

This chapter presents the background required to understand the problem that will be solved by this thesis. Section 2.1 provides the elementary background for analysis of real-time systems used in this report. Section 2.2 describes previous work on federated scheduling including the current state-of-the-art.

2.1 System Model

In this section, the task model that is used in this report is first introduced. After, different types of processor platforms are introduced along with a motivation to why we consider unrelated platforms. Then the role of assignment policies and run-time scheduling policies are explained. Finally, feasibility tests are explained.

2.1.1 Sequential task model

A real-time systems need to schedule a set of n tasks Γ where each task $\tau_i \in \Gamma$ is an independent program. Task τ_i is said to *arrive* when it is ready to execute. A task is defined by a worst-case execution time (WCET) C_i and a deadline D_i which is relative to the tasks time of arrival. Most work considers recurrent tasks where several instances of a task can arrive for a potentially infinite time duration. A recurrent task τ_i can be *periodic*, where the time duration between the arrival of each instance is consistent and defined by its period T_i. A task can also be *sporadic* where the time duration between arrivals is T_i or more. The *utilization* of task τ_i measures how much of a processors capacity is utilized by τ_i, which is defined as C_i/T_i. Finally, a task τ_i can have a deadline that is *implicit* ($D_i = T_i$), *constrained* ($D_i \leq T_i$) or *arbitrary* (D_i is unrelated to T_i).

2.1.2 Multiprocessor platform models

There are three primary distinctions made between different types of multiprocessor platforms.

Firstly, *homogeneous platforms* are systems where each processor is identical to one another. Analysis of such platforms is simpler since the completion times of a task does not depend on which processor it is assigned, but only how many processors a task is assigned.

Secondly, *heterogeneous uniform platforms* allow each processor to execute at different speeds. In a uniform platform all tasks experience the same speedup from

the same processor. This can be seen as identical processors running at different clock frequencies. This means that analyses of these platforms require a speed factor for each processor. Even though this is a common classification of heterogeneous platforms it is limited in its applicability in practice. This is because increasing the clock frequency is unlikely to give an equal speedup on all tasks. One reason for this is that different tasks need to access memory to different extents. This often provides a bottle-neck to the speedup, the extent of which vary between tasks[2].

Finally, in *heterogeneous unrelated platforms* each task runs with an arbitrary execution time on each processor. This means that two tasks may experience different speedups on the same processor. Therefore, each task have a unique WCET for each processor. This makes the analysis of these systems considerably more complicated. One example of an unrelated platform is the ARM big.LITTLE architecture. It is a heterogeneous platform which consist of a set of high performance processors and a set of energy-efficient processors. Even though both processors types are of the same architecture, the speedup provided by the high performance processor varies greatly between different tasks[4] which illustrates the limitation of the uniform platform model. Furthermore, unrelated platforms can model systems with processors of different types and architectures.

In our work we are considering a platform M of m heterogeneous unrelated processors $\mu_x \in M$, since it is a general model which also covers uniform and homogeneous platforms. This gives it a wide applicability but also provides a greater challenge when constructing processor assignment heuristics. Most previous work on federated scheduling covers the case of homogeneous platforms or special cases of heterogeneous platforms. This leaves a significant gap in the literature regarding federated scheduling on unrelated platforms.

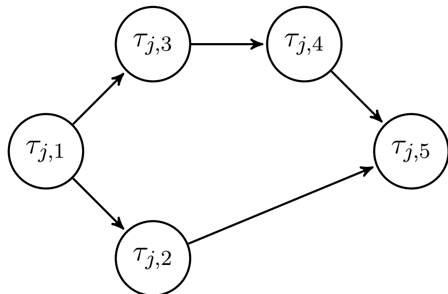
2.1.3 Parallel task model

Many modern applications benefit from internal task parallelism. This is achieved by dividing the task into several parts that can, to an extent, execute in parallel. These partial tasks, referred to as *subtasks* are typically scheduled globally on a set of processors. The Directed Acyclic Graph (DAG) is a powerful model of the internal structure of parallel tasks (see figure 2.1a). Moreover, DAG tasks can be modeled by run-time libraries such as OpenMP[5]. In the DAG model each task is represented as a collection of nodes and edges in a graph. Each node represents a sequential subtask $\tau_{j,i}$ with its own WCET $C_{j,i}$.

Every DAG has a start node, which has no ingoing edges, and an end node, which has no outgoing edges. An edge between two nodes marks a precedence constraint. For example in figure 2.1a subtask $\tau_{j,2}$ and $\tau_{j,3}$ cannot begin its execution before $\tau_{j,1}$ has finished. Thus, all subtask must be scheduled on the available processors in an order that respect the precedence constraints. In a DAG there exists a *critical path* which is the path with the largest total WCET. Thus, the total WCET of the task would be equal to the WCET of the critical path if the task is executed on a platform with an infinite number of available processors. Therefore, the critical path can be seen as a bottle-neck to the completion time of the task.

Note, on unrelated platforms each subtask $\tau_{j,i}$ has a unique WCET $C^x_{j,i}$ for each

processor μ_x. Thus, the WCETs of a parallel task can be represented by a table. An example of such a table can be seen in figure 2.1b. Figure 2.1b shows that each WCET can be entirely arbitrary. Furthermore, the sequential WCET on a particular processor can be provided by the total WCET of the corresponding row. Thus, the DAG model is a generalization of the sequential task model.



(a) Example of task j with 5 subtasks and precedence constraints between them (e.g. $\tau_{j,3}$ precedes $\tau_{j,4}$).

	$\tau_{j,1}$	$\tau_{j,2}$	$\tau_{j,3}$	$\tau_{j,4}$	$\tau_{j,5}$
μ_1	4	2	6	12	9
μ_2	8	5	3	10	2
μ_3	16	25	2	4	3

(b) Example of a table of WCETs for task τ_j in a platform with three processors.

Figure 2.1: Example of a DAG with its corresponding worst-case execution times for each node

2.1.4 Assignment policies

Processor assignment policies determine the manner which tasks and processors are assigned to each other. There are primarily three different ways that assignments can be determined.

Global scheduling is when all tasks share all the processors. This means that tasks are allowed to *migrate* (i.e. move from one processor to another) during runtime. Global scheduling is theoretically the most efficient assignment since any task could be scheduled on any processor. But due to migration, global scheduling need to consider the interference of all other task in the system when calculating the finishing time of a task. The extent of the interference is difficult to analyze offline which leads to an overestimation of the completion times. Hence, global scheduling tend to lead to an overestimation of the required resource.

Partitioned scheduling, on the other hand, assigns each task to specific processors to execute on, which eliminates migration. This reduces the scheduling problem to uniprocessor scheduling on each individual processor. Since analysis of uniprocessor scheduling is a more developed area of research, partitioned scheduling provides better timing guarantees on each individual processor. However, since migration is not possible there is no possibility for tasks running on different processors to share unutilized processor capacity, which tends to result in processor under-utilization. Thus, partitioned scheduling needs to solve the problem of assigning tasks to processors (i.e. the partitioning problem) which is NP-hard.

Federated scheduling is a paradigm that split the tasks into two groups, heavy tasks and light tasks. Light tasks can finish its execution sequentially on a single processor before its deadline. Therefore, a subset of the available processors are

assigned sets of light tasks to execute sequentially without migration. This is equivalent to partitioned scheduling. On the other hand, heavy tasks cannot finish its execution on a single processor as a sequential program, but need to utilize parallelism to finish in time. Such tasks are therefore assigned dedicated sets of processors called clusters, where global scheduling can be used inside the cluster to schedule all the subtasks. Clusters are isolated from one another in the sense that only one parallel task can execute on a cluster. Thus, parallel tasks can not be interfered by other tasks. This allows for more precise analysis within the cluster than can be provided by global scheduling. Federated scheduling is sometimes regarded as a generalization of partitioned scheduling to include tasks that can not meet its deadline when executing sequentially. It has also been shown to be a good alternative to global scheduling since it can provide better timeliness guarantees[6]. Therefore, this work considers federated scheduling which includes the problem of assigning light tasks to a set of processors as well as assigning a cluster of processors to each heavy task (i.e. the partitioning problem and the clustering problem).

2.1.5 Run-time Scheduling Policies

Real-time schedulers needs to determine at which time instant each task should execute. This is most commonly done in run-time using *priorities*. The task with the highest priority gets scheduled first. These priorities are determined by a scheduling policy. Such policies can enforce fixed priorities or dynamic priorities. Fixed priorities are set offline while dynamic priorities are set during run-time by the scheduler. Furthermore, a run-time scheduler may or may not be *preemptive*. A preemptive scheduler will stop a running task whenever a higher prioritized task arrives.

Under global scheduling the run-time scheduler also need to decide which processor the next task should execute on. On a homogeneous system each task is often scheduled on the first available processor. However, on a heterogeneous system which processor a task is scheduled to can have a high impact on its execution time. Therefore, greater care need to be taken by the scheduler when deciding where the tasks should execute.

2.1.6 Feasibility tests

In many real-time applications timing predictability is highly important. This is the ability to determine offline (before run-time) whether a system will meet all its deadlines or not. Within the industry of real-time systems 48% of practitioners describe timing-predictability as very important to their products [1]. For certain industries this number is considerably higher, such as the avionics industry which report up to 78%. One way of achieving timing predictability is by *feasibility test*. These test states whether or not a certain scheduler can schedule a specific task set on a given platform. Feasibility tests are mathematical conditions that are formed from heuristics. This allows system designer to quickly determine the feasibility of a task set without testing each possible schedule.

A feasibility test can be *exact*, in which case it returns 'true' if the task set is feasible and 'false' if it is not feasible. However, since exact analysis of many scheduling

algorithms is unknown, researchers provide pessimistic analysis for such algorithms. As a consequence many feasibility tests are not exact, instead they are often *sufficient*. A sufficient test may return 'false' even when the task set is feasible. In other words, the test provides a condition that is sufficient to guarantee the feasibility of the task set, but a feasible task set does not necessarily satisfy the condition. Such a test is said to be *pessimistic* since it underestimates the capabilities of the scheduler. The pursuit of providing tests that reduces or eliminates the pessimism of different schedulers on varying types of systems is a vast area of research.

Several different techniques are used to provide feasibility tests. Some tests provide an upper-bound on the utilization. As long as the total utilization of the task set does not exceed this bound the task set is feasible. Another method is to analyze the *makespan* of each task. The makespan is the time it takes for a task to finish its execution, which depends on many factors. For instance, the makespan of a sequential task on a non-preemptive scheduler (i.e. the task can not be interrupted) is equal to its execution time. Under a preemptive scheduler the makespan needs to take into account all the possible interference from other tasks. Furthermore, makespan is often applied to determine the feasibility of parallel tasks under global scheduling. However, this requires rigorous analysis that considers the interference between the subtasks in the analyzed task. Furthermore, if the scheduler is preemptive then the interference from other task need to be considered as well. Determining the makespan of a parallel task is an NP-hard problem[7]. Thus, pessimistic upper-bounds of the makespan are used to establish feasibility tests. Such a test states that if, for every task, the upper-bound of the makespan does not exceed the deadline, then the task set is feasible.

2.2 Previous Work

In this section we present some of the most significant findings in the study of real-time systems in general and for federated scheduling in particular.

2.2.1 Uniprocessor scheduling

Real-time scheduling on uniprocessors is a well established area of research going back many years. Perhaps the biggest contribution in this field was by Liu & Layland in 1973 [8]. They introduced the preemptive uniprocessor scheduling algorithms *Rate-Monotonic (RM)* and *Earliest Deadline First (EDF)*. RM is a fixed priority policy where tasks with shorter periods are given higher priorities, while EDF dynamically prioritizes tasks that are closer to their deadline. EDF was shown by [8] to be an optimal priority policy on uniprocessors. Furthermore, they provided an exact utilization-based feasibility test for EDF. This test states that EDF can schedule a task set on a single processor if the total utilization of the tasks is no more than one. On the other hand, RM was shown by [8] to be an optimal fixed-priority scheduler, meaning that no other fixed-priority policy can schedule a task that RM can not. However, RM only has a *sufficient* feasibility test, meaning if the total utilization is lower than $n(2^{1/n}-1)$ then all tasks are guaranteed to meet their deadlines under RM. If the utilization is larger than this bound then feasibility can

not be concluded.

2.2.2 Partitioned scheduling

Partitioned scheduling has been a popular paradigms since it can transfer the rich theoretical framework of uniprocessor scheduling on to multiprocessor systems. However, it needs to solve the problem of assigning tasks to processors. Since this is a bin-packing problem there are several known heuristics that can be used for this allocation problem. One of them is the *First-Fit (FF)* heuristic which assigns each task to the first processor that can feasibly schedule it. Dhall & Liu [9] applied FF to Rate-Monotonic in their *Rate-Monotonic-First-Fit (RMFF)* algorithm. Whether or not the task fits in a processor is determined by the sufficient feasibility test from [8]. Later, Dhall & Liu [10] gave a sufficient utilization-bound feasibility test for RMFF, which states that if the total utilization of the task set is smaller than or equal to $m(2^{1/2} - 1)$ the task set is guaranteed to be feasible by RMFF. However, this bound only allows a total processor utilization of approximately 41% , which is wasteful of processor resources.

In [11] López et al. considered several bin-packing heuristic for EDF on homogeneous platforms. Among them was First-Fit, but also *Best-Fit(BF)* and *Worst-Fit(WF)*. Best-Fit gives each task to the processor with the smallest remaining available utilization that is no smaller than the utilization of the task. On the other hand, Worst-Fit gives each task to the processor with the most available utilization. López et al. provided utilization bounds for each of these heuristics under EDF. They showed that the utilization bound for EDF-FF and EDF-BF were $\frac{\lfloor 1/\alpha \rfloor m+1}{\lfloor 1/\alpha \rfloor+1}$, where α is largest utilization of all the tasks. Additionally, the showed that the utilization bound for EDF-WF is smaller, namely $m - (m-1)\alpha$. Thus, EDF-WF was concluded to be worse than EDF-BF and EDF-FF. Furthermore, they showed that for any possible heuristic the utilization bound can be no greater than $\frac{\lfloor 1/\alpha \rfloor m+1}{\lfloor 1/\alpha \rfloor+1}$ and for any reasonable heuristic the utilization bound can never be smaller than $m - (m-1)\alpha$.

However, Baruah et al. [12] noted that utilization-bounds are flawed when used as a metric to compare the performance partitioned schedulers since it only looks at the total utilization. Therefore, he introduced the *speed-up factor*. The speedup-factor of a scheduler S is how much faster a processor platform need to be if S schedules a task set compared to if an optimal scheduler schedules the task set. Thus, the speedup factor is a measurement of how far a scheduler is from being optimal. Baruah showed that speedup-factor for EDF-BF and EDF-FF is at least $2 - 2/(m+1)$, while the speed-up factor for EDF-WF is at least $2 - 2/m$. Thus, in the worst-case EDF-WF was shown to be closer to an optimal scheduler than EDF-BF and EDF-FF.

2.2.3 Makespan

Scheduling a set of tasks under precedence constraints on a multiprocessor system is a class of problems (known as makespan problems) that have been considered for many years. Many instances of these problems has been shown to be NP-hard [7].

Furthermore, it has long been known that global scheduling a set of tasks on a multi-processor system can give rise to certain types of anomalies. These are unexpected behaviors that occur from changes in the system. For instance, the time it takes to schedule a set of tasks can increase if a processor is added to the platform. This is counter-intuitive yet possible due to the fact that changes in the system can alter which processor each task is scheduled to. In [13] and [14], Graham provided upper-bounds to how much certain anomalies can increase the makespan of a parallel task. He considered a non-preemptive list scheduler which schedules a set of subtasks on a homogeneous platform in a partial order. He showed that, under such a scheduler, the makespan of a parallel task can increase no more than $1 + (m - 1)/m^{'}$ as a consequence of increasing the number of processors from m to $m^{'}$. Similarly, he showed that the equivalent bound $(2 - 1/m)$ with a fixed number of processors m was true when applying changes to the task set. Furthermore, this bounds was shown to be tight, meaning no smaller bounds can be constructed for these changes on similar systems. It follows from this that no non-optimal non-preemptive global DAG scheduler can guarantee that it gives a makespan that is less than $2 - 1/m$ times larger than the makespan provided by an optimal non-preemptive schedule. In addition, in proving the bounds in [13], Graham showed that the makespan of a DAG on a homogeneous system is upper-bounded by $W_j^{\infty} + (W_j^{tot} - W_j^{\infty})/m$ where W_j^{tot} is the sum of the execution times of all tasks and W_j^{∞} is the execution times of all tasks in the critical path. This bound has been used extensively in subsequent work and the list scheduling technique forms the basis for a large range of work that considers the makespan problem on homogeneous platforms[15, 16].

2.2.4 Federated scheduling

Federated scheduling is a relatively new scheduling technique first introduced by Li et al. in [6]. They considered a set of sporadic parallel tasks with implicit deadlines running on a homogeneous platform. Tasks with a utilization greater than one was characterized as heavy tasks, while the remaining tasks where considered as light tasks. Each heavy task was assigned a fixed number of processors while the light tasks were assigned the remaining processors. Li et al. showed that the speedup-factor is 2 for federated scheduling of sporadic tasks with implicit deadlines. In addition, they showed that the speedup factors for *Global Earliest Deadline First (GEDF)* and *Global Rate Monotonic (GRM)* were approximately 2.618 and 3.732 respectively. Federated scheduling was therefore shown to have a better speedup factor establishing it as a good alternative to global scheduling.

Federated scheduling on homogeneous multiprocessor platforms was improved in a few ways. Baruah extended it to arbitrary deadlines [17]. This assignment algorithm assigns a minimum number of processor to a task by estimating its finishing time with the tasks. It was shown that this algorithm has a speed-up factor of $3 - 1/m$. On the other hand, Dihn et. al. [18] created a federated assignment algorithm that creates one possible schedules for the dag while respecting all the precedence constraints. The dag is then allocated $\lceil \frac{C_i}{D_i} \rceil$ processors since this is the least amount of processors needed to schedule the dag. Then, during each iteration of the algorithm a new schedule is created and one processor is also added. The

algorithm then continues to iterate until the dag is schedulable.

Typed DAGs has also been considered for federated scheduling. Han et. al. [19] proposed an algorithm for single typed DAGs. In single typed DAGs every processor is of a certain type and each subtask can only execute on a single type of processor. This algorithm was shown to experimentally outperform GEDF.

Recently, Voudouris et al provided a federated scheduling algorithm that considers heterogeneous unrelated multiprocessors [2]. The scheduler is divided into two parts namely an *inter-task* scheduler and an *intra-task* scheduler. The inter-task scheduler implements an assignment algorithm that assigns clusters of processors to each heavy task while assigning light tasks to single processors. The intra-task scheduler is a runtime scheduler that globally schedules each individual subtask within a heavy task on the assigned cluster. The light tasks are scheduled using EDF on each partitioned processor. The work in [2] is, to the best of our knowledge, the first to provide a federated scheduling algorithm for parallel DAG tasks on unrelated platforms. This thesis build upon [2] by providing an improved processors assignment algorithm (i.e. an inter-task scheduler).

3

Problem Formulation

This work introduces a novel algorithm that solves the processor assignment problem for federated scheduling. Our algorithm is a new federated scheduling algorithm that utilizes the run-time scheduler in [2] to schedule each heavy task on their respective cluster. Similarly, our scheduler use the EDF run-time scheduler from [8] to schedule the light tasks on each partitioned processor. A major limitation of the assignment algorithm in [2] is the fact that it is not social-aware, in the sense that when a task is assigned a processor it does not regard the fact that it is possibly depriving a resource (i.e. processor) from another task. This is problematic since this processor could be crucial for some other task to meet its deadline. Therefore, we provide an assignment algorithm that considers the resource requirement of all tasks. This algorithm utilizes our novel *social-aware* bin-packing heuristic to solve the clustering problem. Furthermore, partitioning is done through a supportive heuristic, meaning each partitioned processor takes on as many tasks as possible.

Our algorithm solves two sub-problems, namely it assigns disjoint sets of light tasks to individual processors and it assigns disjoint sets of processors (i.e. *clusters*) to individual heavy tasks. These two problems will be referred to as the *partitioning problem* and the *clustering problem* respectively. Both problems can be reduced to the classical *bin-packing* problem which is defined as the following: Consider a set of items I that are to be packed into a set of bins B. The size of a subset of items I_x represents how much space the items in the subset occupies inside a bin, and is defined by the function $S(I_x) > 0$. Each bin $b_y \in B$ has a limited amount of space $L_y > 0$. Each item must be packed in some bin that has enough remaining available space. In other words, for each bin $b_y \in B$, the total size of its assigned items I_y must be $S(I_y) \leq L_y$. The bin-packing problem is solved when all items $i \in I$ have been packed in a bin while respecting this condition. An example of a bin-packing problem can be seen in figure 3.1a, where each bin has been assigned some set of items such that their total size never exceed their respective limit. However, the bin-packing problem has been shown to be NP-hard[3], meaning that there exist no general optimal solution that can be computed in polynomial time unless $P = NP$. The optimal solution requires that each possible combination of item-to-bin assignment is tried, which is an exponential-time solution. Since bin-packing is an NP-hard problem, heuristics are required to solve the problem in polynomial-time.

Note, the partitioning problem on a homogeneous platform can be reduced to a bin-packing problem by considering tasks as items and processors as bins, as seen in figure 3.1b. The size of a set of items represents their total utilization. Each processor can be utilized to no more than 100 %, which gives $L_y = 1$ for all processors b_y. Thus, all tasks need to be packed into the available processors such

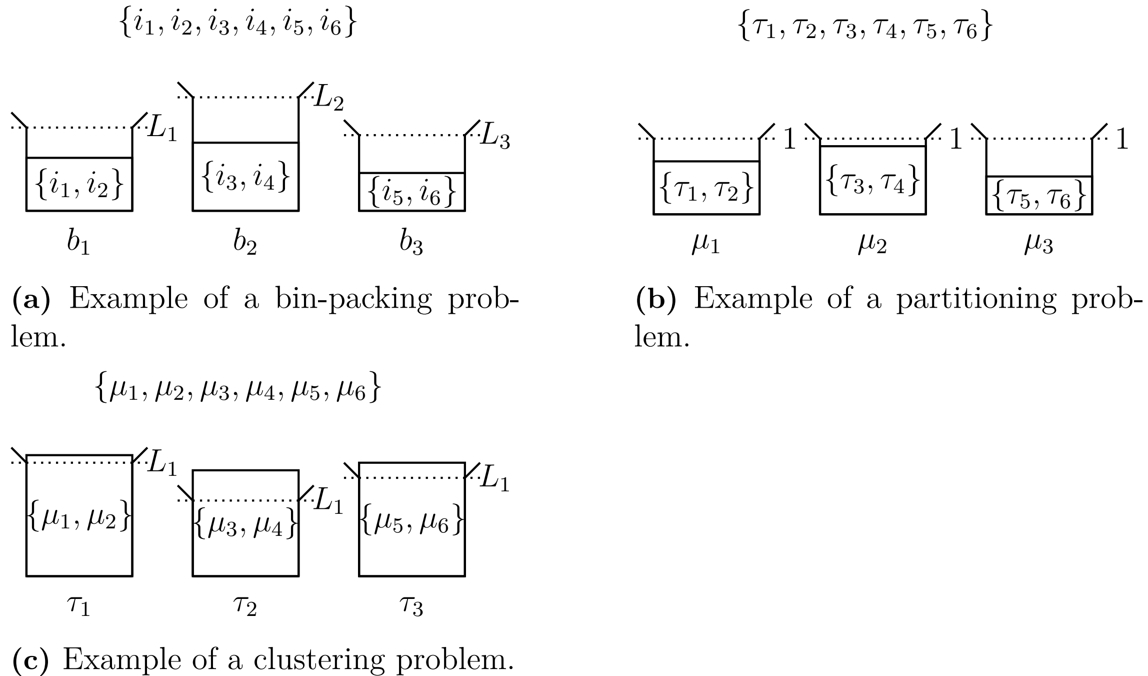
that the total utilization does not exceed 1 on any processor. In other words the condition $S(I_y) \leq 1$ need to be true for all processors b_y. Note, this condition is equivalent to the utilization-based feasibility test for EDF from [8].

In addition to the partitioning problem, federated scheduling also need to solve the clustering problem. The clustering problem on a homogeneous platform can be reduced to a bin-packing problem by representing tasks as bins and processors as items (note, the reverse of partitioning). An example of this can be seen in figure 3.1c. The size of a set of items then measures the processing capacity of the set of processors. The problem can thus be reduced to packing enough processors into each task such that all tasks receive enough processing capacity to meet their deadline. Thus, the limit L_y represents the minimum capacity required by task b_y to meet its deadline. This gives the converse of the classical bin-packing problem since the size of the assigned processors should not be smaller than L_y. This gives the condition $S(I_y) \geq L_y$ for all bins b_y, which is reflected in figure 3.1c by all bins being overfull. This problem is also in general NP-hard. However, unlike partitioning, clustering on homogeneous platforms is a special case of bin-packing that has an optimal solution with a polynomial time complexity[1]. This is because all processors provide the same processing capacity and thus have the same size. This simplifies the problem since it is only relevant how many processors each task gets, but not which ones.

However, when clustering a heterogeneous uniform platform each processor provide different capacities, thus a scheduler needs to consider which processor is assigned to which task. This gives an NP-hard problem. Furthermore, in an unrelated platform each task may execute with different speeds on different processors. Thus, the capacity that a processor is able to provide depends on which task it is assigned to. Meaning, that the clustering problem on unrelated platforms is a generalization of the reversed bin-packing problem. Similarly, partitioning on unrelated platforms is also a generalization of the bin-packing problem. Such a generalization does not increase the time-complexity of the optimal solution, since it is already NP-hard. However, it increases the difficulty of finding good heuristics and adds an additional dimension to the problem. Namely, the size (capacity/utilization) of an item (processor/task) depends on which bin (task/processor) it is assigned. Therefore, the size of any set of items I_x on bin b_y will be denoted $S_y(I_x)$.

We introduce a novel property that can be applied to a bin-packing heuristic. We call this property *social-awareness*. Despite the novelty of this property, it aims to solve a familiar problem. Namely, that some notion of fair distribution of items is desirable, since this increases the likelihood that the bin-packing is successful. Such a fair distribution is different from an equal distribution since fairness entails that processing resources are assigned to the tasks that needs them the most. In this work a social-aware heuristic is developed and applied to the clustering problem for unrelated platforms. Here clustering is considered the main focus of our algorithm since has not been previously explored to the same extent as partitioning. Nonetheless, the heuristic can be equally applied to partitioning as well, with only minor

[1]Even though the clustering problem on homogeneous platforms can be solved optimally in polynomial time in terms of bin-packing, it is not the case for the application of federated real-time scheduling. This is because estimating the capacity is in itself an NP-hard problem in this application.



(a) Example of a bin-packing problem.

(b) Example of a partitioning problem.

(c) Example of a clustering problem.

Figure 3.1: Different resource assignment problems

modifications. Furthermore, the social-aware heuristic can potentially be applied to a wide range of resource allocation problems. In this work we specifically apply it to federated real-time scheduling.

3.1 Limitations

The main contribution of this work is the novel bin-packing heuristic and its application to the federated scheduling problem. Even though we provide a complete federated scheduling algorithm, our contribution is limited to the processor assignment algorithm. The run-time schedulers and associated feasibility tests are contributions made from others, namely Voudouris et al [2] and Liu & Layland [8]. Furthermore, the work only considers task sets with implicit deadlines since this is a necessary condition for the utilization-based feasibility test for EDF from [8]. Furthermore, this work considers sporadic task sets since this is a limitation of the run-time scheduler in [2]. However, note that our assignment algorithm does not provide these limitation which means that the main contributions of this thesis are not limited by these factors.

4

Social-Aware Algorithm

In this chapter our Social-Aware Processor Assignment (SAPA) algorithm is presented. First, the run-time schedulers from [2] and [8] are detailed along with their feasibility tests, since they are used by SAPA . Later, the property of social-awareness is formally defined, explained and motivated in terms of bins and items. Then, this is applied to the clustering problem of real-time tasks on heterogeneous platforms. Finally, the SAPA algorithm is presented, which implements social-aware clustering and supportive partitioning to provide a federated scheduling algorithm.

4.1 Run-time Schedulers

This section details the run-time schedulers and their associated feasibility test used in this work. The entire content of this section consist of contributions made from other work, namely [2] and [8]. Both of these run-time schedulers are the state-of-the-art in their respective field. These run-time schedulers are treated by SAPA as black-boxes, meaning they can easily be replaced by other schedulers that solve the same scheduling problems. However, they are provided here for completeness.

4.1.1 Heavy Tasks

During run-time heavy tasks are scheduled on their assigned cluster in isolation from other tasks. This is equivalent to global scheduling of all the subtasks on the cluster, where each subtask must respect the precedence constraints defined by the DAG. Voudouris et al provides a global scheduler for parallel tasks on unrelated platforms in [2] which is named the Greedy Unrelated Minimum-speed-preference-index (GUM) scheduler. The GUM scheduler has a *greediness* property which ensures that subtasks will always migrate to a faster processor whenever one is available. The algorithm maintains two queues, *ReadyQ* and *RunQ*. ReadyQ stores all the subtasks that has not started executing yet while RunQ stores the currently executing subtasks. Furthermore, a list of speeds is stored for each subtask $\tau_{j,i}$. Each list contains the speed $\delta^x_{j,i}$ of a subtask $\tau_{j,i}$ for each processor μ_x in descending order. The speed $\delta^x_{j,i}$ is defined as $C^{min}_{j,i}/C^x_{j,i}$ where $C^{min}_{j,i}$ is the WCET of subtask $\tau_{j,i}$ on its fastest processor. Thus, the processor which provides the ith speed entry in the list will be referred to as the ith *speed-preference* of that subtask.

The GUM scheduler performs a new scheduling decision whenever the parallel task arrives or one of the subtasks completes. It is only in these instances of time that new processors will become available. The algorithm updates the schedule by

first iterating through the RunQ. It will find the subtask with the biggest speed-preference for any of the idle processors. The selected task will migrate to its preferred processor, thus making its current processor idle. The RunQ is continuously iterated until no more of the running subtasks can migrate to a faster processor. If there still remains processors that are idle and the ReadyQ is not empty, then the first subtask is removed from the ReadyQ and dispatched to its most preferred idle processor. This is done continuously until either all processors are occupied or until the ReadyQ is empty. The schedule will remain until the next subtask completes which triggers a new scheduling decision.

Associated with the GUM scheduler is the feasibility test in Theorem 1 which determine if the scheduler can guarantee feasibility on its given cluster. It is based on an upper-bound of the makespan since finding the exact makespan is an NP-hard problem. Furthermore, throughout this thesis, whenever the makespan is computed this bound is used.

Theorem 1 (Corollary 2 from [2]). *An upper bound of the makespan of τ_j scheduled with GUM on an unrelated platform M_j with m processors is given by equation 4.1. Task τ_j is feasible on M_j if the upper-bound does not exceed the deadline of τ_j.*

$$\chi^j_{M_j} \leq \frac{W^{tot}_{j,M_j} + (m-1) \cdot W^{\infty}_{j,M_j}}{\zeta_{j,M_j}} \leq D_j \tag{4.1}$$

An upper-bound on the makespan $\chi^j_{M_j}$ of τ_j when executing on its assigned cluster M_j is computed in equation 4.1. It relies on the notion of *workload* which represents the amount of work the platform needs to perform to complete the task. The workload is divided by the capacity of the platform to compute the makespan. However, each processor provides a different capacity to different tasks in an unrelated platform. Therefore, equation 4.1 introduces the *the minimum capacity* ζ_{j,M_j} that the cluster M_j can provide to tasks τ_j. Note, the greediness property ensures that for each speed-preference i there is at least one processor that executes at a speed no slower than its i^{ith} speed. In other words, in the worst case, one subtask executes on the processor that provides its first speed-preference, one subtask executes on the processor that provides its second speed-preference etc. Therefore, the minimum capacity of the platform is provided when processors execute at the slowest speed for each speed-preference. Thus, a lower-bound of the capacity is the sum of the slowest speed for each speed-preference. This lower-bound captures the heterogeneity of the platform.

The *total workload* W^{tot}_{j,M_j} is the workload that a platform would experience if all the subtasks were executed sequentially on its fastest processor. If subtasks were independent the total workload could be divided by the minimum capacity to compute an upper-bound on the makespan. However, since the task is a DAG with precedence constraints there may be time instances where some processors are idle even though there are unexecuted subtasks. This can be regarded as the processors doing additional idle work (i.e. work that does not belong to any subtask). Thus, the *pseudo-workload* is defined as the work that these idle processors would have performed if they weren't idle. To find an upper-bound on the pseudo-workload equation 4.1 use the workload of the critical path W^{∞}_{j,M_j}. The critical path is defined

as the path with the largest workload if all subtasks execute on their preferred processor. Since, in the worst case, all but one processor is idle during execution of the critical path, an upper bound on the pseudo-workload is $(m-1) \cdot W^{\infty}_{j,M_j}$, where m is the number of processors in the cluster. Thus, the upper-bound of the makespan is defined as the total workload and the pseudo-workload divided by the minimum capacity. Finally, a sufficient feasibility test is established by checking that this upper-bound does not exceed the deadline.

4.1.2 Light Tasks

The partitioning in our assignment algorithm assumes an EDF run-time scheduler which was introduced by Liu & Layland in [8] and its feasability test is presented in Theorem 2. Such a scheduler has a ReadyQ of sequential tasks that have arrived but are yet to be executed on the processor. Whenever the ReadyQ is not empty and the processor is idle the first task in the queue is dispatched on the processor. The order in the ReadyQ is determined by the priorities of the tasks which are dynamic. Whenever a task is added or removed from the ReadyQ it is sorted in ascending order with respect to the *absolute deadline*. The absolute deadline of a task τ_j is its relative deadline D added to the time instant which the task arrived. The task in the ReadyQ with the shortest absolute deadline is dispatched on the processor.

Theorem 2 (Theorem 7 from [8]). *For a given set of tasks Γ_x that are assigned to processor μ_x, the EDF scheduling algorithm is feasible if an only if*

$$U_{\Gamma_x} = \sum_{\tau_j \in \Gamma_x} \frac{\sum_{\tau_{j,i} \in \tau_j} C^x_{j,i}}{T_j} \leq 1 \tag{4.2}$$

Theorem 2 checks that the total *utilization* of all tasks Γ_x that are assigned to processor μ_x are not larger than 1. The utilization is defined as the WCET of the task divided by its period. However, in this work each task is represented as a DAG with multiple WCETs per node. Thus, the sequential WCET of a DAG task on μ_x is the sum of all subtasks WCETs on μ_x.

4.2 Social-Awareness

Social-awareness aims to provide a fair distribution of items to each bin. If one considers an assignment algorithm that assigns items one by one, then this algorithm should assign each item to the bin that gives the most fair outcome. For partitioning this means to give the item to the bin that is least burdened by it. For clustering this means to assign the item to the bin that needs it the most. The need that a certain bin has for a certain item can be quantified by measuring the cost of removing that item from the bin's assigned set. Similarly, for partitioning the burden of a certain item can be quantified by measuring the cost of adding the item to the bin's assigned set. In order to compute these costs we introduce the *Augmentation Cost Function*. A social-aware clustering algorithm should assign each item to the bin with the largest augmentation cost. In contrast, a social-aware partitioning algorithm should

assign each item to the bin with the smallest cost. Since the algorithm provided in this work is only social-aware with respect to clustering, definition 1 defines the Augmentation Cost Function for a clustering algorithm. Furthermore, figure 4.1 provides an illustration of all the elements used in equation 4.3. The corresponding definition for partitioning can be found in appendix A.1.

Definition 1. *The Augmentation Cost Function (ACF) for clustering defines the cost of a certain augmentation of the items assigned to bin b_j from I_a to I_b where $I_b \subset I_a$*

$$ACF_{clust}(b_j, I_a, I_b) = \frac{\frac{S_j(I_a) - S_j(I_b)}{S_j(I_a)}}{\frac{S_j(I_b) - L_j}{L_j}} \tag{4.3}$$

$$S_j(I_b) > L_j \tag{4.4}$$

For this definition, assume a clustering problem where a bin b_j is already assigned some set of items I_a such that the total size $S_j(I_a)$ is larger than the limit L_j, i.e. the bin is overfull[1]. The Augmentation Cost Function (ACF) provides a measurement of how a certain augmentation (i.e. change) to the assigned set of items affects b_j. Namely, to what extent does removing a set of items from b_j reduce the total size assigned to b_j. Thus, I_b is the set of items assigned to b_j after the augmentation. The set difference between $I_a \setminus I_b$ is referred to as the *augmentation set*. Thus, ACF measures how bin b_j is effected when removing the augmentation set from I_a.

In equation 4.3, $S_j(I_a) - S_j(I_b)$ gives the size reduction that removing the augmentation set provides. When this size reduction is divided by the size of the originally assigned set I_a the *contribution* of the augmentation set is provided. The contribution therefore measures how much the augmentation set contributes to total size of the assigned items I_a. This contribution is further divided by the *relative augmentation distance*, i.e. the relative distance between the size of the new assigned set I_a and the limit L_j. The distance $S_j(I_b) - L_j$ measures how close the bin is to reaching it's limit after the augmentation, and is denoted as d in figure 4.1. This distance is relative to the limit itself through a division by L_j. This ensures that the ACF is comparable across different bins, since each bin has different limits. Note, when the contribution of the augmentation set increases, so does the ACF. This reflects that the cost of losing the augmentation set is higher if it provides a large part of the total size. In addition, when the relative augmentation distance is decreased the ACF is increased. This reflects the fact that the closer the bin is to its limit, after the augmentation is performed, the higher the cost of removing the augmentation set.

However, note that equation 4.3 makes the assumption that I_b (i.e. the assigned set after the augmentation) has a size that is greater than L_j. This is due to the fact that if $S_j(I_b)$ is smaller than L_j it means that bin b_j will fail if the augmentation is performed. However, such an augmentation would give a negative ACF if it was

[1]Note, $S_j(I_a) > L_j$ is not a necessary condition for ACF, however this assumption is made here for illustration purposes

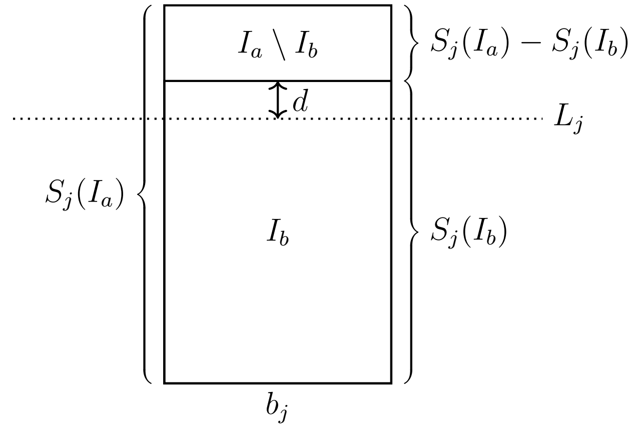


Figure 4.1: Diagram of the elements of the Augmented Cost Function for clustering

allowed, which does not reflect the fact that the bin would fail. If I_b is exactly L_j the ACF gives a division by zero which must be avoided. Therefore, ACF is undefined when $S_j(I_b)$ is smaller than or equal to L_j.

ACF provides the foundation of social-awareness since it is used to determine which bin should be assigned a certain item. However, note that which items are included in I_a and I_b is not clearly defined other than that I_a includes the augmentation set and I_b does not. Thus, social-awareness rely on the notion of *assigned sets* and *potential sets*. The assigned set I_j of a bin b_j is the set of items that is assigned to b_j. The potential set $I_j^{'}$ is the set of all items that can potentially be assigned to b_j. Hence, $I_j^{'}$ includes the assigned set I_j but also includes all the yet unassigned items in I. A social-aware algorithm use the potential set when computing the ACF such that $I_a = I_j^{'}$. Furthermore, it assigns items one by one, meaning that the augmentation set contains one item each iteration. The property of social-awareness for a clustering algorithm is defined by definition 3.

Definition 2. *Consider a clustering algorithm A that assigns a set of items I to a set of bins B. For each iteration k an item $i_x \in I$ is assigned to some bin in $B^k = \{b_j | b_j \in B \wedge S_j(I_j^k) < L_j\}$ where I_j^k is the items already assigned to b_j in iterations leading up to k. The augmentation cost $\phi_x^{j,k}$ of removing item i_x from b_j's potential set $I_j^{'k}$ is defined as $ACF_{clust}(b_j,\ I_j^{'k},\ I_j^{'k} \setminus \{i_x\})$. A is **social-aware** iff during no iteration k it assigns item i_x to a bin $b_j \in B^k$ where $\phi_x^{j,k}$ is defined, if there exist some other bin $b_i \in B^k \wedge b_i \neq b_j$ with an undefined $\phi_x^{i,k}$, or with a defined $\phi_x^{i,k}$ such that $\phi_x^{j,k} < \phi_x^{i,k}$.*

Definition 2 states that a social-aware clustering heuristic gives each item one-by-one to some bin that is not already full (B^k). If $B^k = \emptyset$ for any iteration k then the algorithm has succeeded since there is no bin that needs any more items. This definition requires that the ACF is used to define the cost $\phi_x^{j,k}$ of removing i_x from the potential set $I_j^{'k}$ of b_j in iteration[2] k. The definition states that each item must go to the bin with the largest cost, unless there exists one or more bins

[2]Note, $I_j^{'k}$ depends on k since I_j^k contains all the items that have been assigned to b_j in iterations leading up to k. However, for the remainder of this thesis k will be omitted from the notation

with an undefined cost. In such a case the algorithm must assign the item to one of the bins with an undefined cost. However, which of those bins may be arbitrarily chosen. Thus, each item is assigned to bin that needs it the most. If one or more bins reach their limit when loosing a specific item, then the item is given to one of those bins. Since social-awareness use potential sets, the cost is quantified while taking into consideration the processing capacities of all the available items as well as the already assigned items.

Note, definition 2 defines social-awareness for a clustering algorithm. Meaning, this definition does not allow for partitioning. Hence, a federated scheduling algorithm will not be able to adhere to definition 2. Therefore, in definition 3 social-awareness is made slightly less strict for algorithms that solve clustering and partitioning. For this we introduce a function $P^k(b_j)$.

$$P^k(b_j) = \begin{cases} true, & b_j \text{ can not be clustered for any iteration } i \geq k \\ false, & otherwise \end{cases} \tag{4.5}$$

Function $P^k(b_j)$ asserts that bin b_j can only be considered for partitioning from iteration k and beyond. This decision must be defined by the algorithm since the determining factors may be highly specific to the application. However, if $P^k(b_j)$ returns true for some iteration k then $P^i(b_j)$ must return true for any iteration $i > k$. $P^k(b_j)$ is used in definition 3 to limit which bins need to be considered for partitioning.

Definition 3. *Consider an algorithm A that solves clustering and partitioning by assigning items I to bins B. For each clustering iteration k an item $i_x \in I$ is assigned to some bin in $B^k = \{b_j | b_j \in B \wedge S_j(I_j^k) < L_j \wedge \neg P^k(b_j)\}$ where I_j^k is the items already assigned to b_j in iterations leading up to k. The augmentation cost $\phi_x^{j,k}$ of removing item i_x from b_j's potential set $I_j'^k$ is defined as $ACF_{clust}(b_j, I_j'^k, I_j'^k \setminus \{i_x\})$. A is* ***social-aware with respect to clustering*** *iff during no clustering iteration k it assigns item i_x to a bin $b_j \in B^k$ where $\phi_x^{j,k}$ is defined, if there exist some other bin $b_i \in B^k \wedge b_i \neq b_j$ with an undefined $\phi_x^{i,k}$, or with a defined $\phi_x^{i,k}$ such that $\phi_x^{j,k} < \phi_x^{i,k}$.*

In this definition a clustering iteration is any iteration that considers all the bins that are not yet full and not yet decidedly partition bins. During such an iteration i_x must go to the bin with the largest cost (or undefined) cost. This means that any bin b_p that is decidedly a partitioned bin in clustering iteration k can not receive item i_x, regardless of its cost $\phi_x^{p,k}$. Thus, the only difference from definition 2 is which bins are considered in each iteration. However, this is not an insignificant modification since it depends greatly on the definition of $P^k(b_j)$. For instance, if $P^k(b_j)$ is defined to always return *true* then any arbitrary Algorithm A will have no clustering iterations meaning it is effectively not solving the clustering problem, in which case definition 3 is meaningless. If $P^k(b_j)$ always returns *false* then it is not solving the partitioning problem and definition 3 is equivalent to definition 2.

since costs are never compared across iterations, making the notation obsolete.

Hence, definition 3 provides the flexibility necessary to apply social-awareness to a federated scheduling algorithm.

4.2.1 Social-aware clustering

Definition 2 provides the defining property of a social-aware clustering algorithm in the general case. This need to be applied to the clustering problem found in federated scheduling of real-time tasks. This is done by translating the bin-packing terminology in definition 2 to their real-time scheduling equivalents. This translation gives an equation that is equivalent to equation 4.3. Since definition 3 only considers social-awareness with respect to clustering, the translation can be used to apply definition 3 to a federated scheduling algorithm as well.

Recall that the clustering problem for a federated scheduler represents processors as items and tasks as bins. Thus, the size $S_j(I)$ represents the capacity that the set of processors I provides to task b_j. Defining the capacity of a processor is not a trivial problem since the capacity depends on many things such as clock frequency, memory hierarchies and much more. Furthermore, defining the capacity based solely on characteristics of the processor neglects how well a task may be able to utilize the capabilities of the processor. For instance, a processor with many cores will provide more processing power to a parallel task than a sequential task, since sequential tasks can only utilize one core. However, note that when considering real-time systems, we specifically want to know how long it takes for a task to finish, i.e. the makespan. Therefore, a processor that provides a high capacity to task τ_j should give a short makespan on τ_j. Hence, the capacity provided by μ_x to τ_j can be defined as the inverse of the makespan $\chi^j_{\mu_x}$ of τ_j when executing on μ_x. Furthermore, this definition can be equally applied to a set of processors. Meaning, the capacity that platform M provides to τ_j is defined as the inverse of the makespan χ^j_M. Thus, the size (i.e. capacity) of a set of processors I is equivalent[3] to the inverse of the makespan as shown in equation 4.6.

$$S_j(I) \leftrightarrow \frac{1}{\chi^j_I} \tag{4.6}$$

Recall that definition 3 considers potential sets of items. These represent *potential clusters* M^*_j that include all the processors assigned to task τ_j as well as all the yet unassigned processors in M. The capacity of a potential cluster M^*_j is therefore equivalent to the size of a potential set I'_j as shown in equation 4.7

$$S_j(I'_j) \leftrightarrow \frac{1}{\chi^j_{M^*_j}} \tag{4.7}$$

Furthermore, the limit L_j represent the capacity required by τ_j to meet its deadline. Just like the makespan, the deadline is a measurement of time. Therefore, the limit L_j is equivalent to the inverse of the deadline D_j as shown in equation 4.8.

[3]Note, by equivalent we are referring to a *representational* relationship, which we denote with the symbol $\leftrightarrow$. Meaning, $A \leftrightarrow B$ *iff* A represents B and vice versa. Such a representational relationship allows A to be replaced by B and B to be replaced by A.

$$L_j \leftrightarrow \frac{1}{D_j} \tag{4.8}$$

A makespan-based ACF can be derived using equation 4.7 and 4.8 by replacing the size and the limit with the inverse makespan and the inverse deadline respectively. This ACF is named the benefit function β^j_{x,M^*_j} and measures the benefit of assigning processor μ_x to τ_j when it has the potential cluster M^*_j. This benefit function will be used by the social-aware algorithm to determine which task is assigned processor μ_x.

$$\beta^j_{x,M^*_j} = \begin{cases} \dfrac{\chi^j_{M^*_j \setminus \{\mu_x\}} - \chi^j_{M^*_j}}{D_j - \chi^j_{M^*_j \setminus \{\mu_x\}}}, & \text{if } D_j > \chi^j_{M^*_j \setminus \{\mu_x\}} \\ \infty, & \text{otherwise} \end{cases} \tag{4.9}$$

The following lemma asserts that the benefit function in equation 4.9 is equivalent to the ACF in equation 4.3 whenever condition 4.4 is true.

Lemma 1. *The benefit equation β^j_{x,M^*_j} is equivalent to the Augmentation Cost Function $ACF_{clust}(b_j, I'_j, I'_j \setminus \{i_x\})$ when $S_j(I_b) > L_j$ (condition 4.4) is true, given that $B \leftrightarrow \Gamma$, $I \leftrightarrow M$, $i_x \leftrightarrow \mu_x$, $b_j \leftrightarrow \tau_j$ and $I'_j \leftrightarrow M^*_j$*

Proof.

$$ACF(b_j, I'_j, I'_j \setminus \{i_x\}) = \frac{\dfrac{S_j(I'_j) - S_j(I'_j \setminus \{i_x\})}{S_j(I'_j)}}{\dfrac{S_j(I'_j \setminus \{i_x\}) - L_j}{L_j}} \tag{4.10}$$

equation 4.7 and 4.8 gives:

$$\begin{aligned} &\frac{\frac{\frac{1}{\chi^j_{M_j^*}} - \frac{1}{\chi^j_{M_j^*\setminus\{\mu_x\}}}}{\frac{1}{\chi^j_{M_j^*}}}}{\frac{\frac{1}{\chi^j_{M_j^*\setminus\{\mu_x\}}} - \frac{1}{D_j}}{\frac{1}{D_j}}} \\ \leftrightarrow\ &= \frac{(\frac{1}{\chi^j_{M_j^*}} - \frac{1}{\chi^j_{M_j^*\setminus\{\mu_x\}}}) \cdot \chi^j_{M_j^*}}{(\frac{1}{\chi^j_{M_j^*\setminus\{\mu_x\}}} - \frac{1}{D_j}) \cdot D_j} \\ &= \frac{(\frac{\chi^j_{M_j^*\setminus\{\mu_x\}} - \chi^j_{M_j^*}}{\chi^j_{M_j^*} \cdot \chi^j_{M_j^*\setminus\{\mu_x\}}}) \cdot \chi^j_{M_j^*}}{(\frac{D_j - \chi^j_{M_j^*\setminus\{\mu_x\}}}{\chi^j_{M_j^*\setminus\{\mu_x\}} \cdot D_j}) \cdot D_j} \\ &= \frac{\chi^j_{M_j^*\setminus\{\mu_x\}} - \chi^j_{M_j^*}}{D_j - \chi^j_{M_j^*\setminus\{\mu_x\}}} \end{aligned} \tag{4.11}$$

From equation 4.7 and 4.8 we get the following equivalence for condition 4.4:

$$S_j(I_b) > L_j \leftrightarrow \chi^j_{M_j^*\setminus\{\mu_x\}} < D_j \tag{4.12}$$

Thus when condition 4.12 is true then equation 4.10 is equivalent to equation 4.9. □

Lemma 1 states that the benefit equation is equivalent to the ACF when condition 4.12 is met. However, ACF is undefined when the condition is not met. Therefore, the benefit is set to infinity in this case. This reflects the fact that the cost of removing the processor is infinitely high since it would cause the task to fail (i.e. miss its deadline). This means that if there are several tasks that computes an infinite benefit value for the same processor, which of the tasks that receives the processor is not defined by social-awareness. This means that an algorithm that gives each processor to the task with the highest benefit is social-aware as defined by definition 3. Furthermore, Lemma 2 illustrates an important attribute of the benefit function.

Lemma 2. $\beta^j_{x,M_j^*} < \beta^i_{x,M_i^*}$ *implies* $\phi^j_x < \phi^i_x$ *or that* ϕ^i_x *is undefined and* ϕ^j_x *is defined.*

Proof. For β^i_{x,M_i^*} there are two cases.
(Case $\beta^i_{x,M_i^*} < \infty$):
$\beta^i_{x,M_i^*} < \infty \rightarrow \chi^i_{M_i^*\setminus\{\mu_x\}} < D_i$. Furthermore, $\beta^i_{x,M_i^*} > \beta^j_{x,M_j^*} \rightarrow \beta^j_{x,M_j^*} < \infty \rightarrow \chi^j_{M_j^*\setminus\{\mu_x\}} < D_j$. Then, by lemma 1 we have $\beta^i_{x,M_i^*} \leftrightarrow \phi^i_x$ and $\beta^j_{x,M_j^*} \leftrightarrow \phi^j_x$. Thus, $\beta^j_{x,M_j^*} < \beta^i_{x,M_i^*}$ implies $\phi^j_x < \phi^i_x$
(Case $\beta^i_{x,M_i^*} = \infty$):

$\beta^i_{x,M^*_i} = \infty \rightarrow \chi^i_{M^*_i \setminus \{\mu_x\}} \geq D_i \rightarrow \phi^i_x$ is undefined. Furthermore, $\beta^i_{x,M^*_i} = \infty > \beta^j_{x,M^*_j} \rightarrow \chi^j_{M^*_j \setminus \{\mu_x\}} < D_j \rightarrow \phi^j_x$ is defined. □

Lemma 2 states that if task τ_i has a larger benefit than task τ_k, then τ_i has either a larger augmentation cost than τ_k or τ_i has an undefined augmentation cost while τ_k has not. In either case social-awareness mandates that processor μ_x is not given to τ_k. Therefore, a clustering algorithm for federated scheduling is made social-aware by assigning each processor to the task with the highest benefit value with arbitrarily broken ties.

4.2.2 Supportive partitioning

The partitioning heuristic provided in this work is not social-aware as defined by definition 3. Instead, it takes on a supportive role, meaning each processor that is available for partitioning tries to relieve the other processors of as much workload as possible. Therefore, a partitioned processor takes tasks one by one until no more tasks can fit inside it. Whether or not a task fits in the processor is determined by the EDF feasibility test in Theorem 2. Tasks are taken by the processor in ascending order of *relative utilization*. The relative utilization $\xi^j_{x,M}$ is the utilization of τ_j on μ_x relative to utilization of all processors in M. Thus, the order that processor μ_x takes tasks $\tau_j \in \Gamma$ is determined by equation 4.13. This order will create a preference for items that have a small utilization on processor μ_x but a generally large utilization on all other processors. Thus, μ_x aims to take away a large workload from other processors while taking on a small workload for itself, which allows it take more tasks.

$$\xi^j_{x,M} = \frac{U^j_x}{\sum_{\mu_y \in M} U^j_y} \tag{4.13}$$

4.3 Social-Aware Algorithm

The *Social-Aware Processor Assignment (SAPA)* algorithm is presented in algorithm 1, which inputs a task set and a set of processors, and outputs the allocations final allocations. This algorithm primarily focuses on the clustering problem. This problem is solved using the following heuristic. The platform is iterated through one processor at a time. The algorithm will give each processor to the task that receives the largest benefit from it, as defined by equation 4.9. Whenever a task has received enough processors to pass the feasibility test in Theorem 1 it is removed from the task set. If a task is feasible after receiving only a single processor, supportive partitioning adds additional tasks to that processor. Algorithm 1 includes two sub-algorithms, *Cluster* and *Partition* found in algorithm 2 and algorithm 3 respectively. Algorithm 2 assigns a processor μ_x to some task in Γ. Algorithm 3 assigns tasks in Γ to processor μ_x.

Algorithm 1 takes as input a task set Γ and a set of processors M. It returns the partitions and clusters for each processor and task respectively. In addition it returns *success* if an assignment, that guarantee the feasibility of Γ on M, could

be found. Otherwise it returns *failure*. Algorithm 1 iterates as long as there are processors left to allocate (line 4). In each iteration a processor μ_x is removed from the platform (line 5). Then, μ_x is assigned to a task in Γ by calling algorithm 2 (line 6). Algorithm 2 will update the platform and the task set as well as the clusters and partitions. If the updated task set doesn't have any more tasks to schedule, then the assignment is successful (line 7). However, if this does not occur before the algorithm runs out of processors then the assignment has failed.

Algorithm 2 assigns processor μ_x to some task in Γ and returns the updated clusters and partitions. It begins by constructing a potential cluster M_j^* for each task τ_j based on their previously assigned clusters in M_{clust} and all the remaining processors in M (line 2). Next, the task set is sorted with respect to the benefit of processor μ_x in descending order (line 4). The sorted task set is stored in Γ_s. Then, the first element in Γ_s is stored in τ_a since it is the task that receives the greatest benefit from μ_x (line 5). Recall however that the benefit will be infinite for any task that is infeasible without μ_x. This means that there may be several tasks with an infinite benefit. Therefore, each task that has an infinite benefit is stored in an additional list Γ_f (line 8). If this list contains more than one task it is sorted in ascending order with respect to the relative utilization using equation 4.13 (line 11). Here the relative utilization provides a back-up metric when the benefit is no longer useful for discriminating between tasks. Then τ_a is updated to contain the failing task with the lowest relative utilization (line 12).

If τ_a so far has not been assigned any processors then μ_x will be its first (line 14). Then, the exact feasibility test in Theorem 2 can be used to see if τ_a is a light task for processor μ_x (line 15). If the test is passed τ_a is added to the partition of μ_x and removed from the task set (line 16 - 17). Furthermore, since μ_x is a uniprocessor, additional tasks can be assigned to it by calling algorithm 3 (line 18). This provides the supportive partitioning, meaning that tasks that can execute on μ_x together with τ_a can be removed from the task set. This is done as soon as μ_x has been established as a uniprocessor so that these tasks do not need to be considered when assigning any of the following processors. However, if the test does not pass then τ_a is a heavy task for μ_x. Then μ_x is added to the cluster of τ_a and the task remains in the task set so it can receive more processors (line 20).

If M_a has previously been assigned processors, then τ_a is again considered a heavy task. Then μ_x is assigned to the cluster of τ_a and the feasibility test provided by Theorem 1 is used to determine feasibility of τ_a (line 23 - 24). If the test passes, τ_a is removed from the task set since it don't need any additional processors (line 25). If the test does not pass the task simply remains in the task set.

Algorithm 3 provides supportive partitioning by removing tasks from the task set Γ and assigning them to processor μ_x. The purpose is to relieve the processors in M from the workloads of the tasks in Γ as much as possible by assigning them to μ_x. Therefore, the tasks in Γ are sorted with respect to their relative utilizations $\xi_{x,M}$ in ascending order (line 1). The sorted list Γ_s is iterated through in order (line 2). For each task τ_j in the ordered set the feasibility test in Theorem 2 is applied (line 3). The test is applied while considering τ_j and the tasks Γ_x^{part} already partitioned to μ_x. If the test passes τ_j is added to Γ_x^{part} and removed from Γ (line 4-5). Furthermore, the cluster that had already been assigned to τ_j is returned to the platform (line 6).

Algorithm 1: Social-Aware Processor Assignment (SAPA)

input : $\Gamma = \{\tau_1, \tau_2, \tau_3, ..., \tau_n\}$,
$M = \{\mu_1, \mu_2, \mu_3, ..., \mu_m\}$
output: $\Gamma_{part} = \{\Gamma_1^{part}, \Gamma_2^{part}, \Gamma_3^{part}, ..., \Gamma_m^{part}\}$,
$M_{clust} = \{M_1^{clust}, M_2^{clust}, M_3^{clust}, ..., M_n^{clust}\}$,
Status: *success/failure*

```
1  ∀Γ_i ∈ Γ_part : Γ_i ← ∅;
2  ∀M_i ∈ M_clust : M_i ← ∅;
3  Status ← failure;
4  while M ≠ ∅ do
5      μ_x ← M.pop();
6      {M, Γ, M_clust, Γ_part} ← Cluster(μ_x, M, Γ, M_clust, Γ_part);
7      if Γ = ∅ then
8          Status ← success;
9          break;
10     end
11 end
```

Theorem 3. *Algorithm 1 is a Social-Aware clustering algorithm as defined by definition 3, where tasks represents bins* ($B \leftrightarrow \Gamma$) *and processors represent items* ($I \leftrightarrow M$) *and* $P^k(\tau_j) = \tau_j \in \bigcup_{i=1}^{m} \Gamma_{i,k}^{part}$ *where* $\Gamma_{i,k}^{part}$ *is the current value of* Γ_i^{part} *during iteration* k.

Proof. Here we denote Γ^k as being the real-time scheduling equivalent to B^k. Then, from lemma 2 and definition 3 we know that a federated scheduling algorithm is social-aware if and only if for every clustering iteration k it gives μ_x to some task in Γ^k and it never gives μ_x to $\tau_j \in \Gamma^k$ when there is some other task $\tau_i \in \Gamma^k$ where $\beta_{x,M_j^*}^j < \beta_{x,M_i^*}^i$. This hold true for Algorithm 1 due to the following.

First we prove that μ_x is always given to a task in Γ^k. Note, each iteration k is an iteration in the loop in line 4 to 11 in Algorithm 1, since one processor is assigned in each such iteration. Lets assume by contradiction that during some iteration k, μ_x is assigned to $\tau_p \notin \Gamma^k$. This implies that τ_p is in a partition ($P^k(\tau_p) = true$) or that $\chi_{M_p}^p \leq D_j$ (equivalent to $S_p(I_p) \geq L_p$). However, if $P^k(\tau_p) = true$ then τ_p has been added to a partition in some iteration $i < k$. This implies that τ_p has been removed from Γ in iteration i (line 17 in Algorithm 2 and line 5 in Algorithm 5). Hence, τ_p can not be in Γ during iteration k. This means that μ_x can not have been assigned to task τ_p in iteration k, which is a contradiction. Furthermore, if τ_p is in Γ^k then τ_p has failed the test in either Theorem 1 or Theorem 2 after being assigned a processor in all iteration $i < k$, since otherwise it would have been removed from Γ (line 15 and line 24 in algorithm 2). However, note that Theorem 1 is more pessimistic than Theorem 2, meaning $\chi_{\{\mu_x\}}^p \leq D_j \rightarrow U_x^{\{\tau_p\}} \leq 1$. Therefore, if $\chi_{\{\mu_x\}}^p \leq D_j$ in iteration k then τ_p must have passed one of the tests in some iteration i when it was assigned a processor, which is a contradiction. Hence, it is proven that Algorithm 1 always assign i_x to a task in Γ^k in every iteration k.

Now we prove that Algorithm 1 never gives μ_x to $\tau_j \in \Gamma^k$ when there is some other

Algorithm 2: Cluster($\mu_x, M, \Gamma, M_{clust}, \Gamma_{part}$)

1 **foreach** $M_j^{clust} \in M_{clust}$ **do**
2 $\quad M_j^* \leftarrow M \cup M_j^{clust}$; // assign temporary platforms
3 **end**
4 $\Gamma_s \leftarrow Sort(\Gamma, \beta_{x,M^*})$;
5 $\tau_a \leftarrow \Gamma_s.head()$;
6 $\Gamma_f \leftarrow \emptyset$;
7 **foreach** $\tau_j \in \Gamma_s \wedge \beta^j_{x,M_j^*} = \infty$ **do**
8 $\quad \Gamma_f \leftarrow \Gamma_f \cup \{\tau_j\}$
9 **end**
10 **if** $|\Gamma_f| > 1$ **then**
11 $\quad \Gamma_f \leftarrow Sort(\Gamma_f, \frac{1}{\xi_{x,M}})$;
12 $\quad \tau_a \leftarrow \Gamma_f.head()$;
13 **end**
14 **if** $M_a^{clust} = \emptyset$ **then**
$\quad$ // Test if τ_a is feasible using Theorem 2
15 $\quad$ **if** $U^x_{\{\tau_a\}} \leq 1$ **then**
16 $\quad\quad \Gamma_x^{part} \leftarrow \{\tau_a\}$; // Assign τ_a to μ_x
17 $\quad\quad \Gamma \leftarrow \Gamma \setminus \{\tau_a\}$;
18 $\quad\quad \{M, \Gamma, M_{clust}, \Gamma_{part}\} \leftarrow Partition(\mu_x, M, \Gamma, M_{clust}, \Gamma_{part})$;
19 $\quad$ **else**
20 $\quad\quad M_a^{clust} \leftarrow M_a^{clust} \cup \{\mu_x\}$; // Assign μ_x to τ_a
21 $\quad$ **end**
22 **else**
23 $\quad M_a^{clust} \leftarrow M_a^{clust} \cup \{\mu_x\}$; // Assign μ_x to τ_a
$\quad$ // Test if τ_a is feasible using Theorem 1
24 $\quad$ **if** $\chi^a_{M_a} \leq D_a$ **then**
25 $\quad\quad \Gamma \leftarrow \Gamma \setminus \{\tau_a\}$;
26 $\quad$ **end**
27 **end**
28 **return** $\{M, \Gamma, M_{clust}, \Gamma_{part}\}$;

Algorithm 3: Partition($\mu_x, M, \Gamma, M_{clust}, \Gamma_{part}$)

```
1  Γ_s ← Sort(Γ, 1/ξ_{x,M});
   // add tasks to μ_x until feasibility test fails
2  foreach τ_j ∈ Γ_s do
3      if U_{Γ_x^{part} ∪ {τ_j}} ≤ 1 then
4          Γ_x^{part} ← Γ_x^{part} ∪ {τ_j};
5          Γ ← Γ \ {τ_j};
6          M ← M ∪ M_j^{clust};
7          M_j^{clust} ← ∅;
8  end
9  return {M, Γ, M_clust, Γ_part};
```

task $\tau_i \in \Gamma^k$ where $\beta^j_{x,M^*_j} < \beta^i_{x,M^*_i}$ during iteration k. Algorithm 1 calls Algorithm 2 once for every processor μ_x in M. In line 20 in Algorithm 2 μ_x is assigned to τ_a. Note, τ_a is either the head of Γ_s or the head of Γ_f. If τ_a is the head of Γ_s then, from the sort in line 4 we know that there is no other task τ_i in Γ in iteration k where $\beta^a_{x,M^*_a} < \beta^i_{x,M^*_i}$. If τ_a is the head of Γ_f then $\beta^a_{x,M^*_a} = \infty$, meaning there can be no τ_i where $\beta^a_{x,M^*_a} < \beta^i_{x,M^*_i}$. □

Thus, SAPA is shown to be social-aware with respect to the clustering problem. Note, Algorithm 1 is primarily a clustering algorithm. However, Supportive partitioning can be seen as an optimization in this context since it decreases the tasks under consideration for future iterations. Importantly, a processor is only partitioned after it has been assigned to a task by the clustering heuristic. Furthermore, it only removes tasks that it can successfully partition in to this processor. Therefore, supportive partitioning does not take any resources from the clustering problem, only workloads. Hence, the integrity of social-aware clustering is preserved.

Recall, when several tasks have undefined ACF (i.e. infinite benefit) definition 3 does not dictate which of them should receive the processor. However, one could make the argument that if more than one task fails without the processor then the algorithm should fail since only one task can receive it. This would be a correct assessment if we were only solving the clustering problem. However, the federated assignment problem also encompasses the partitioning problem. Due to the pessimism in the makespan feasibility test in Theorem 1, a task that fails the makespan test may yet still be feasible on a single processor. Hence, failing the algorithm in the case of multiple infinite benefit values would be undesirable. Furthermore, the makespan feasibility test in equation 1 is not anomaly-free, meaning the makespan may decrease if a processor is removed from the tasks cluster. This means that a task that gives an infinite benefit at a certain point may actually give a benefit that is not infinite in a future iteration when processors are more scarce. This is another reason why the algorithm should not fail at the presence of infinite benefits. However, due to these complexities determining which of these task should receive the processor is not trivial. Our solution use the relative utilization from equation 4.13 to determine this.

4.3.1 Algorithmic complexity of SAPA

Algorithm 1 has a polynomial time complexity. The algorithm iterates over all m processors. However, each partitioned task gives back its assigned cluster to the platform. Since each task can be partitioned at most once there can be at most $m \cdot n$ iterations. In each iteration Algorithm 2 is called once. In each execution of Algorithm 2, Algorithm 3 can be called at most once. Thus, the time complexities of Algorithm 2 and Algorithm 3 are upper-bounded by the execution time of the $Sort()$. A sort function can be implemented with $\mathcal{O}(n \log n)$ time-complexity and since only tasks are sorted the input to $Sort()$ is upper-bounded by n. Furthermore, each comparison done by $Sort()$ needs to compute either the makespan or the relative utilization. We denote complexity of of computing the makespan and the relative utilization as $\mathcal{C}_\chi$ and $\mathcal{C}_\xi$ respectively. Thus, the computational complexity of Algorithm 1 is bounded by $\mathcal{O}(m \cdot n^2 \cdot \log n \cdot \max\{\mathcal{C}_\chi, \mathcal{C}_\xi\})$. Since $\mathcal{C}_\chi$ and $\mathcal{C}_\xi$ are also computed in polynomial-time, Algorithm 1 is computed in polynomial-time.

5 Simulation

5.1 Experimental Setup

A simulator is used to test each algorithm. This simulator runs one of the algorithms for a specified number of iterations. Each iteration the algorithm is executed with a randomly generated model of the platform and the task set. The executed algorithm will return success or failure for each iteration. The acceptance-ratio is thus the number of times the algorithm returned success compared to the total amount of iterations. The simulator is called for each algorithm in order to compare their respective acceptance-ratios. Moreover, the algorithms simulate on the same random task sets. This ensures that the comparisons between them are fair.

In order to properly evaluate the algorithm, the simulator provides several parameters to influence the type of system that it generates. The number of processors is determined by the parameter ($P \geq 1$). Furthermore, the heterogeneity parameter ($0 \leq H \leq 1$) determines how many different types of processors there are relative to the total number of processors. A value of zero gives a homogeneous system while a value of one gives a system where each processor is of a different type. In addition, the uniformity parameter ($0 < U \leq 1$) decides how similar the speedup every subtask experience from each processor. A value of one means that the platform is entirely uniform while a value close to zero means that there can be great variation between the speed of each subtask for the same processor type.

All tasks are limited to having no more than ($ST \geq 1$) subtasks. In many applications a task may consist of identical subtasks operating on different data, referred to as Single Program Multiple Data (SPMD). Therefore, subtasks can be split into different subtask types, where each subtask within a type has the same WCET. The parameter ($0 \leq V \leq 1$) defines the maximum variety between subtasks within a task. A value of zero forces all tasks to be of a single type while a value of one allows all subtasks to be unique. Additionally, a deadline multiplier ($DM \geq 1$) is provided to influence the largest possible deadline of a task. With a small deadline multiplier the simulator is more likely to generate heavy tasks while with a large deadline multiplier it is more likely to produce light tasks. Tasks are generated one by one based on these parameters. The utilization of a generated task τ_j is computed as w^j/D_j. Tasks are generated until the total utilization reaches or surpasses the goal utilization ($0 < G \leq 1$). If the goal utilization was surpassed the deadline of the final task is adjusted to achieve the desired total utilization. The goal utilization thus determine what percentage of the platform need to be utilized to meet all deadlines.

5.1.1 Generating platform

Before any algorithm can be simulated the model of the system need to be generated. First the platform is generated by computing the number of processor types $PT = max\{1, \lfloor P \cdot H \rfloor\}$. The distribution of the processors over all the processor types is determined using the *integer_UUnifast* algorithm (see appendix C)[1], which is a modified algorithm based on UUnifast presented in [20]. This algorithm provides an array of size PT where the values of each element is the number of processors in the respective type, and the sum of all elements is equal to P. After the distribution of processor types has been established, each processor μ_x is assigned a slow-down factor $f_x = rand(1, PT \cdot 2 + 1)$. This factor is later used to compute the WCETs of the subtasks.

5.1.2 Generating DAGs

The simulated DAG model consist of a table of all the WCETs. There is a WCET for all subtask types and processor types. Furthermore, two arrays are stored. One array stores how many subtasks there are in each subtask type. The second array stores how many subtasks of each subtask types that are in the critical path. In addition, the number of task types, the total number of subtasks and the deadline is stored. This data is generated and then used to compute the benefit values as well as the makespan and utilization.

Each tasks is generated in the following way. First, the number of subtasks in the task is generated by selecting a random value in the range $[1, ST]$. Then the number of subtask types (TT) is determined by selecting a random value within the range $[1, ST \cdot V]$. subsequently, the number of subtasks in each subtask type is determined by the *integer_UUnifast* algorithm. It gives an array with TT number of elements where each element of the array determines the number of subtasks in their respective type, and the sum of all the elements is equal to ST. Then an initial WCET is generated for each subtask type by selecting a random value in the range of $[10, 500]$. Following, for each subtask type a new WCET is generated for each processor type by multiplying the initial WCET with a random value in the range of $[f_x \cdot U, f_x/U]$. The total workload (W^{tot}_{j,M_j}) is calculated by summing the multiplication between the number of subtasks in each type and the initial lowest WCET. W^{∞}_{j,M_j} is calculated by taking a random amount of subtasks from each type and multiplying this amount by the initial lowest WCET and summing these values together. The deadline is selected to be a random value in the range $[\chi^j_p \cdot DM, w^j \cdot DM]$ where w^j is the *inflated workload*. This is the total workload $W^{tot}_{,}$, divided by the minimum capacity $\zeta_{,}$. Thus w^j inflates the workload such that is representative of a heterogeneous platform.

5.1.3 Comparisons

The acceptance ratio provided by SAPA is compared to the inter-task scheduler in [2], which will be referred to as the *Selfish Processor Assignment (SPA)* algorithm.

[1] In contrast to the original $UUnifast$ algorithm, the $integer_UUnifast$ algorithm ensures that all elements are integer values greater than or equal to one.

This algorithm gives each task its preferred cluster of processors without any regard for the other tasks. When a task is assigned its preferred cluster those processors are remove from the processor set. Thus, the next task is assigned its preferred cluster from all the remaining processors. A task will prefer the cluster with the highest *cluster value*. The cluster value is the sum of the *processor values* of all processors in the cluster. The processor value measures how valuable a processor is to a certain task and therefore serves the same purpose as the benefit value provided in this work. Thus, we also measure the performance of SPA where we replace the original processor value with the benefit function from equation 4.9. This way the performance contributions made by the benefit equation can be isolated from the contributions made by the SAPA algorithm itself.

Furthermore, we define two additional processor value equations that aims to capture the value of a processor in more detail than the original. We denote these variations as *PV-speed* and *PV-workload*. PV-speed base the processor value on the speeds that the processor provides to the task, while PV-workload additionally gives a weight to each task based on their workload.These equations are further described in appendix A.2.2 and A.2.3 respectively. Moreover, these processor values have been made "fair" as described in appendix A.2.4. Thus, besides the SAPA algorithm, we compare the performance of SPA using the makespan-based benefit function, original processor value, *fair PV*, *fair PV-speed* and *fair PV-workload*.

Note, the simulator is not specifically designed to exclusively generate feasible task sets. Therefore, when an algorithm fails it does not necessarily reflect poor performance since the task set may have been impossible to schedule. Hence, measuring the performance of SAPA in isolation would not provide meaningful insight. Therefore, the performance of SAPA is always compared to the performance of SPA as a reference.

5.2 Simulation Results

In this section we show how the performance of SAPA compares to that of SPA and its variations. Namely, we compare the performance of

- **SAPA**: Social-Aware Processor Assignment.
- **SPA original PV**: Current state-of-the-art proposed by [2].
- **SPA makespan**: modified [2] where processor value is replaced with eq. 4.9.
- **SPA fair PV**: modified [2] with a "fair" processor value (see appendix A.2.4).
- **SPA fair speed**: modified [2] with a "fair" processor value based on speed (see appendix A.2.2).
- **SPA fair workload**: modified [2] with a "fair" processor value based on workload (see appendix A.2.3).

First, general performance is presented as acceptance-ratio plotted to increasing goal utilization. The performances are first shown for heterogeneous platforms with different uniformity (U), deadline mulitplier (DM) and subtask variance (V). Then we show the performance on homogeneous platforms with different deadline multipliers. The figures shown in this sections are extracts from the more comprehensive set of graphs found in appendix B.

5.2.1 General Performance

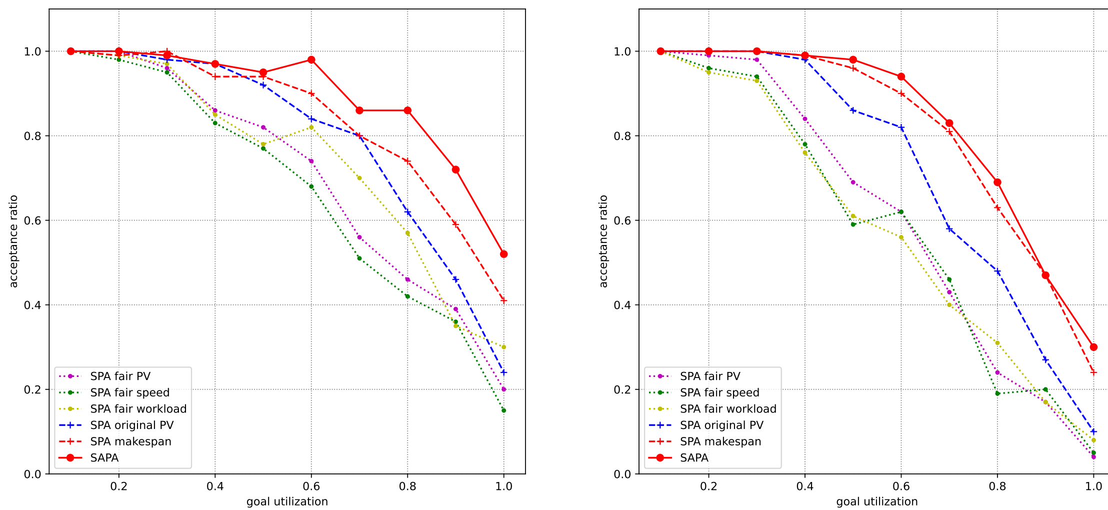
Figure 5.1 shows the performance of each algorithm running on a platform with 66% heterogeneity (H=0.66). Each subplot present the cases where the platform has a uniformity factor of 0, 0.5 and 1 respectively. The x-axis spans a goal utilization between 0.1 and 1. The y-axis shows the acceptance ratio of each algorithm variation. Notably, SAPA outperforms all the variations of the SPA algorithm. The most significant increase can be seen in figure 5.1c when the goal utilization is 60% (G=0.6). Here SAPA provides an acceptance ratio more than double that of the original SPA.

Note, a large part of the performance increase can be directly attributable to the makespan benefit function. This is illustrated by the increased acceptance-ratio in SPA makespan compared to SPA original in figure 5.1, which is particularly noticeable on uniform platforms. Thus, equation 4.9 is shown to be effective even when used in an algorithm that is not social-aware. This is likely due to the fact that it provides a value that is directly related to the feasibility test in Theorem 1, since it is based on makespan. On the other hand, the original processor value is disconnected from the feasibility test since it is merely based on the speeds provided by the processors. This ignores the DAG structure of the tasks entirely. For instance, a processor may provide very fast speeds to all subtasks within a task, except the subtasks in its critical path. Then the processor may not increase the likelihood that the task is feasibility yet it would still provide a high processor value. However, the lack of improved feasibility would be reflected by a marginal change in the makespan. Thus equation 4.9 considers the DAG structure to the same extent as the upper-bound of the makespan in equation 4.1. This direct relationship between the benefit function and the makespan provides, in most cases, a better estimation of the value of a processor.

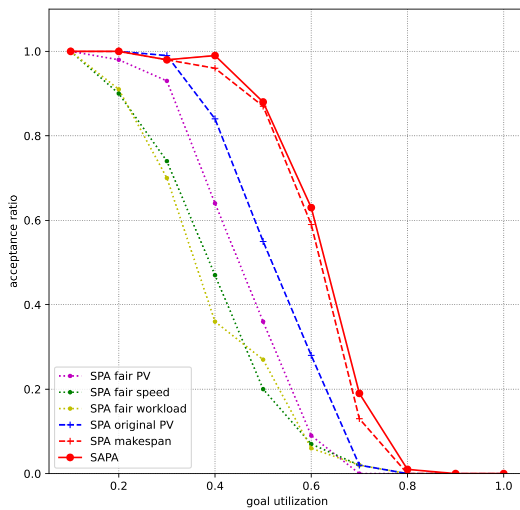
Note, as the uniformity increases the curve more closely resembles an S-curve. This is likely a property of the simulator since all algorithms essentially follow the same curve shape. Figure 5.1a and 5.1b shows the beginning of an S-curve with a generally higher acceptance ratio. This reflects the fact that less uniform platforms are more likely to be feasible. This is because there is less competition between tasks since they are more likely to prefer different processors.

5.2.2 Changing the number of heavy tasks

The deadline multiplier is used to change the utilization of each individual task. A small deadline multiplier gives a task set with few heavy tasks, while a large deadline multiplier gives a task set with many light task. Figure 5.2 shows the performance of SAPA, SPA original and SPA makespan when running unrelated platforms with varying deadline multipliers (DM). As shown by figure 5.2b and 5.2c, SAPA provides a large performance increase when the deadline multiplier is increased. This indicates that supportive partitioning plays an important role in providing good assignments. This is likely due to the fact that supportive partitioning fills each partitioned processor with tasks in order to utilize the processor as much as possible. This philosophy is absent in the SPA algorithm. Additionally, a contributing factor could also be that the supportive partitioning sorts the tasks



(a) U = 0 **(b)** U = 0.5

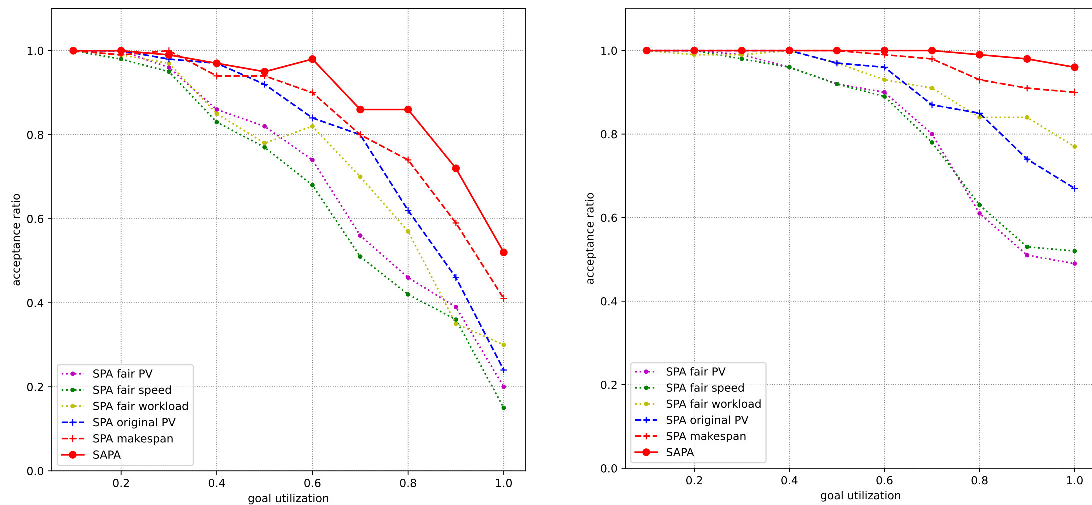


(c) U = 1

Figure 5.1: Acceptance-ratio of all algorithm variations under a platform with 66% heterogeneity and varying uniformity, plotted against goal utilization

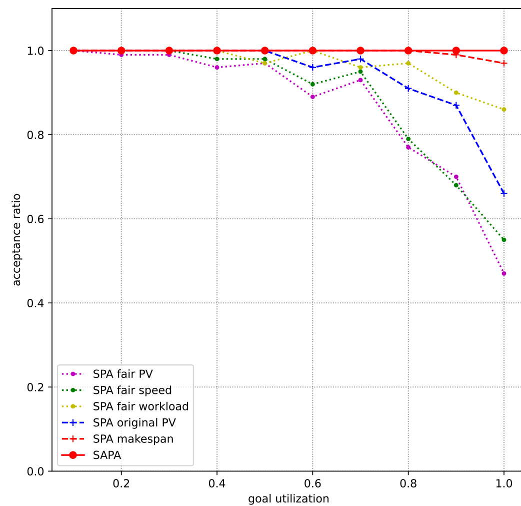
based on their relative utilization. This is likely quite effective since the utilization is directly related to the optimal EDF feasibility test.

Furthermore, note that the higher the deadline multiplier the higher acceptance ratio is for all algorithms. This is likely in part due to the simulator. Namely, the goal utilization is measured as the sum of inflated workloads w^j divided by D_j for all tasks τ_j. However, when tasks are almost exclusively partitioned the real utilization of each task will only consider the processor that it is executing on. Thus, when most tasks are partitioned the real utilization would likely be smaller than the goal utilization. Therefore, it is not unreasonable that the acceptance ratio remains at 100% even with a 100% goal utilization as seen in figure 5.2c



(a) DM = 1

(b) DM = 2

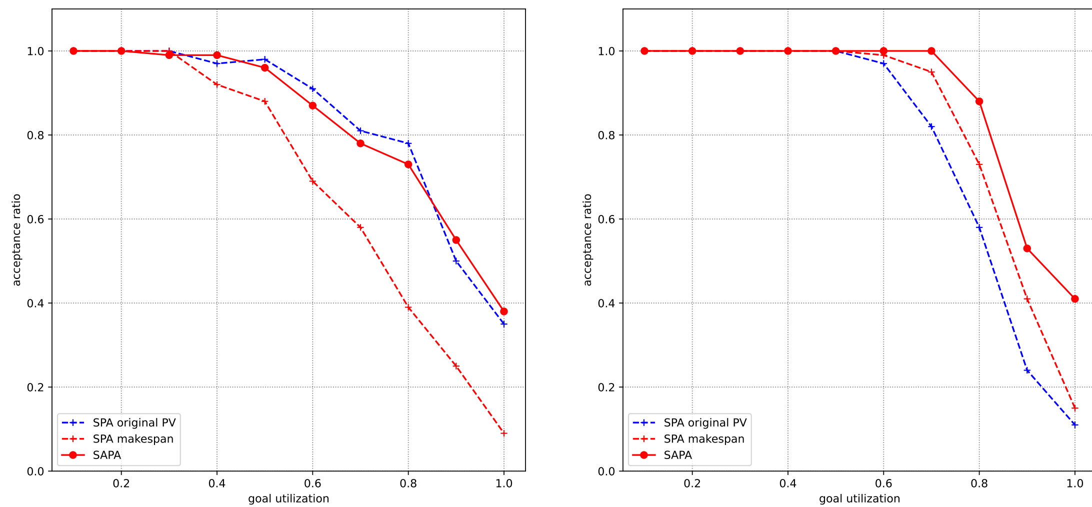


(c) DM = 3

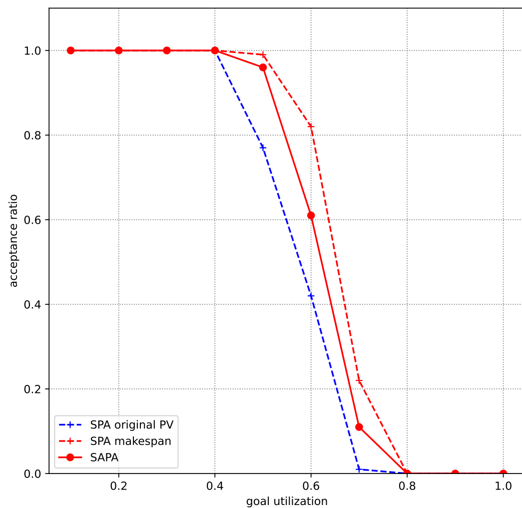
Figure 5.2: Acceptance-ratio on an unrelated platform with varying deadline multipliers

5.2.3 Changing the number of processors

Although SAPA mostly outperforms SPA original, this is not always the case. When the number of processors is increased to 32 there seem to be cases where SAPA performs worse than SPA. This is illustrated by figure 5.3 which shows how the algorithms perform on 32 processors. Here SPA original seems to perform better than SAPA on highly unrelated platforms, i.e. platforms where the uniformity is 10% (U=0.1). This can be seen in figure 5.3a. However, this does not persist when the uniformity is increased as seen in figure 5.3b and 5.3c. In fact, figure 5.3c shows that SPA makespan performs better than SAPA on a uniform platform. Thus, the uniformity seems to have a significant impact on the performances of the SPA algorithm. This impact seems to be exacerbated by increasing the number of processors.



(a) U = 0.1 **(b)** U = 0.5

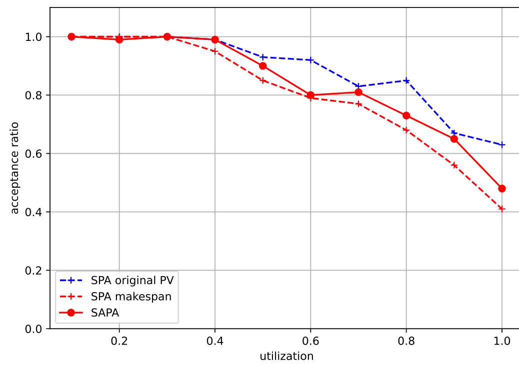


(c) U = 1

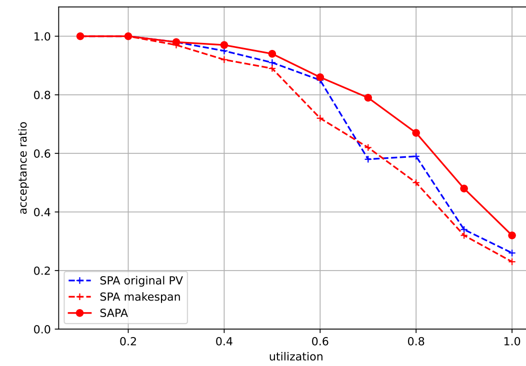
Figure 5.3: Acceptance-ratio with 32 processors with varying uniformity

5.2.4 Changing the subtask variance

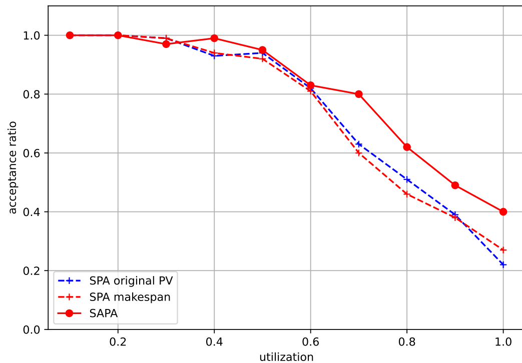
Another instance where SPA original seems to be performing better than SAPA is when tasks with no subtask variance execute on a highly unrelated platform. This can be seen in figure 5.4a. However, this is not the case when the subtask variance is increased, as seen in figure 5.4b and 5.4c. In fact, the subtask variance does not seem to have a clear effect on the acceptance-ratio except for in the case where it is zero. This suggests that SPA original may be a better alternative for applications using SPMD tasks on highly unrelated platforms. However, when the subtask variance is more than zero SAPA still seem to outperform SPA. Furthermore, when the uniformity is increased SAPA also outperforms SPA as seen in appendix B.4.



(a) V = 0



(b) V = 0.5



(c) V = 1

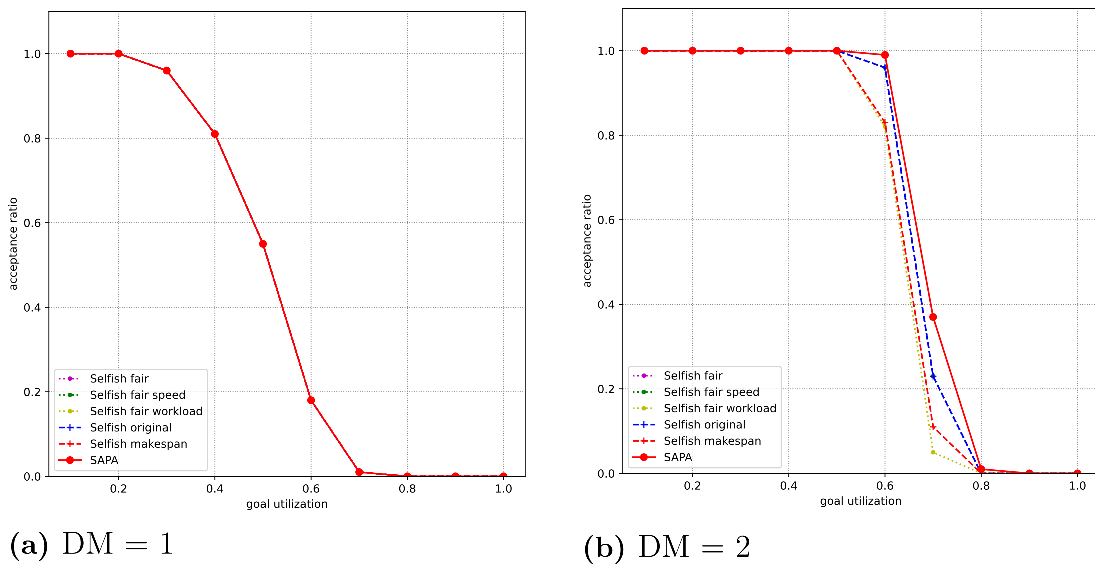
Figure 5.4: Acceptance-ratio of task sets with different node variance executing on a highly unrelated platform

5.2.5 Homogeneous platforms

Figure 5.5a shows that all algorithms performs identically on homogeneous platforms, which is somewhat expected. Recall that the clustering problem can be solved optimally (with respect to a non-optimal test) on homogeneous platforms in polynomial time. This is because all processors are identical, thus each task needs some number of processors to meet their deadline. Both SPA and SAPA provide an optimal solution for the homogeneous case since no task is given more processors than it needs and the algorithm does not fail until all processors are spent. Meaning, both algorithms will give each task exactly as many processors it needs to pass the test unless it runs out of processors. However, SAPA and SPA solves partitioning differently, yet they still have identical performance. This is due to the fact that the deadline multiplier is set to one where the simulator tends to generate exclusively heavy tasks. This means that partitioning is essentially not possible in figure 5.5a. However, in figure 5.5b the deadline multiplier is set to two which generates a task set that mostly consist of light tasks that can be partitioned. In this case SAPA slightly outperforms all the variations of SPA. Interestingly, the original processor value outperforms the makespan-based benefit function when used on SPA. This indicate that the performance increase provided by SAPA is due to the supportive partitioning and not the makespan-benefit function. This is not surprising since the

makespan benefit function is optimized for clustering but not partitioning.

It may seem counter-intuitive that the benefit function should have any effect on a homogeneous platform since all processors are identical and thus should provide identical benefits. However, the benefit function does determine the cluster-value of each tasks respective cluster. Thus, the benefit function influences the order in which SPA selects tasks. Seemingly the task order provided by the original processor value gives better results than the task order provided by equation 4.9 in this case. This could be due to presence of anomalies in the makespan equation, or merely be a consequence of the fact that makespan equation is not adapted to partitioning.



(a) DM = 1 **(b)** DM = 2

Figure 5.5: Acceptance-ratio of all algorithms under a homogeneous platform

6

Discussion and Conclusion

The social-aware clustering heuristic is introduced to provide a sense of fairness when assigning resources. Such a fair distribution differs from an equal distribution since it gives each item to the bin that needs it the most. This is a desirable attribute since different bins may need different amount of resources. Furthermore, different bins may not be able to utilize each item to the same extent, as is the case for federated scheduling on heterogeneous platforms. Hence, social-awareness is defined to be highly generalized such that it does not assume that the size of an item is fixed. Thus, the size of an item may depend on which bin it is assigned. Additionally, the size is measured for entire sets of items, meaning, the size contribution of a single item may depend on the rest of the set. These generalizations allow for the application of the social-aware heuristic to the clustering problem in federated scheduling of real-time tasks on unrelated platforms. Furthermore, such generalizations have, to the best of our knowledge, never been provided by any other bin-packing heuristic that aims to distribute items fairly across bins.

In this thesis the social-aware heuristic was first defined for the clustering problem (Definition 2). This definition relies on the novel Augmentation Cost Function (ACF) which measures the cost of changing which items are assigned to a bin. Then, the definition of social-awareness was slightly modified to consider algorithms that solves clustering as well as partitioning (Definition 3). This modification allows social-awareness to be applied to the federated real-time scheduling problem. Thus, the benefit equation, which is based on makespan, applies the ACF to the real-time scheduling domain. Furthermore, the novel SAPA algorithm was introduced to solve the assignment problems for federated scheduling on unrelated platforms. This algorithm was shown to be social-aware with respect to clustering, since it gives each processor to the task with the highest benefit. Finally, through simulation SAPA was shown to mostly perform better than the current state-of-the-art in [2], referred to as SPA.

The following sections discusses the limitations of this work and how they may be addressed by future work. Furthermore, we discuss the applicability of social-awareness to other bin-packing problems.

6.1 Simulation Results

As shown in section 5.2 SAPA mostly outperforms SPA. However, there are cases where SAPA performs worse than SPA. Most noticeably is that the performance of SAPA performs worse as the number of processors increase. However, it could also

be the case that SPA performs particularly well when the number of processors are increased. The reason for this is yet not known. Furthermore, due to the increasing execution time of the algorithms, larger numbers of processors than 32 has not been tested. In addition, SPA performed better in the specific case where there is no subtask variance and very low uniformity. Why this is the case has not been established.

6.2 Effects of Anomalies

SAPA does not require an anomaly-free[1] makespan equation. This is because processors are allocated offline, meaning anomalies can not occur during run-time. Thus, a successful assignment will never lead to a missed deadline. Furthermore, the makespan will decrease when an anomaly occurs, which leads to a negative benefit. Naturally, this means that the task that experience an anomaly from the processor will be sorted after all the other tasks, which seems desirable. However, there are reasons to believe that anomalies may give less desirable processor assignments. Namely, anomalies are transitory in nature, meaning that if the makespan of a certain task decrease by removing a processor from a platform, then it does not mean that removing the same processor from a different cluster will also decrease the makespan. In other words, even though a processor may provide a negative benefit to task τ_j under cluster M_A does not mean it will provide a negative benefit to τ_j under cluster M_B. Thus, the benefit may fluctuate between positive and negative values, meaning it may not capture the true value of a processor. Furthermore, logs from the simulation showed that anomalies were commonly occurring, thus it likely has a noticeable impact on performance. Therefore, replacing equation 4.1 with an anomaly-free upper-bound of the makespan would likely give better processor assignments.

6.3 Processor Order

Definition 3 does not enforce any particular order that processors should be considered. Thus, Algorithm 1 considers an arbitrary processor order. However, the order that processors are considered may have considerable impact on the performance. For a uniform platform the processor could be define as the fastest processors first or the slowest processors first. However, on an unrelated platform there may not be a single fastest processor. There are other ways that the processor order can be determined. One solution would be to compute the benefits of each processor for each task before assigning any of the processors. The order may then be defined by the total benefit of each processor, in ascending or descending order. However, the processor order was not explored in this thesis. The simulator generates the set of processors in ascending order of the speed factor f_x. Meaning, in a uniform platform

[1]Note, here we are specifically talking about the anomaly that removing a processor may decrease the makespan. Other anomalies such as decreasing the WCET of a subtask increases the makespan, can invalidate SAPA. Fortunately, equation 4.1 is absent of such anomalies.

this would give a processor order of decreasing speed. The effects of changing the processor order has not been tested in simulation.

6.4 Applicability

Definition 3 defines the property of social-awareness with respect to a clustering algorithm. Thus any other resource allocation problem that is equivalent to the clustering problem could apply a social-aware assignment algorithm. The necessary component is a quantifiable metric that correspond to the size of items. In this thesis the item size represent the inverse of the makespan. Furthermore, there needs to be a constraint that correspond to the limit. This constraint must be of the same unit as the metric that correspond to the size. In this thesis the limit represents the inverse of the deadline. Since makespan and deadline have the same unit (time) they are comparable and can be applied to the ACF. Other resource allocation problems may have other constraints. Additionally, in appendix A.1 we provide the definition of social-awareness for the partitioning problem. Since the partitioning problem is equivalent to the original bin-packing problem (i.e. not reversed), this opens for wide range of applicable problems.

6.5 Future Work

The simulation results illustrate that SAPA does not outperform SPA in all cases. These cases need to be better understood. The problem could lie in the clustering algorithm, the partitioning algorithm or the interaction between them. In order to investigate this, social-awareness could be applied to pure clustering problems and pure partitioning problems. This would give a clearer understanding of the merits of the social-aware heuristics. Furthermore, the processor order may be an important aspect of the heuristic that warrants further exploration. Finally, anomalies are believed to have a poor impact on SAPA. Therefore, finding an anomaly-free upper-bound of makespan on unrelated platforms would relevant future work.

Bibliography

[1] Benny Akesson et al. "An Empirical Survey-based Study into Industry Practice in Real-time Systems". In: *2020 IEEE Real-Time Systems Symposium (RTSS)*. 2020, pp. 3–11. DOI: 10.1109/RTSS49844.2020.00012.

[2] Petros Voudouris, Per Stenström, and Risat Pathan. "Federated Scheduling of Sporadic DAGs on Unrelated Multiprocessors". In: *ACM Trans. Embed. Comput. Syst.* 20.5s (2021). ISSN: 1539-9087. DOI: 10.1145/3477018. URL: https://doi.org/10.1145/3477018.

[3] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990. ISBN: 0716710455.

[4] Chao Chen et al. "A Task Scheduling Algorithm Based on Big.LITTLE Architecture in Cloud Computing". In: *2020 6th International Conference on Big Data and Information Analytics (BigDIA)*. 2020, pp. 94–99. DOI: 10.1109/BigDIA51454.2020.00023.

[5] Roberto Vargas, Eduardo Quinones, and Andrea Marongiu. "OpenMP and timing predictability: A possible union?" In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2015, pp. 617–620. DOI: 10.7873/DATE.2015.0778.

[6] Jing Li et al. "Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks". In: *2014 26th Euromicro Conference on Real-Time Systems*. 2014, pp. 85–96. DOI: 10.1109/ECRTS.2014.23.

[7] Jeffrey D Ullman. "NP-complete scheduling problems". In: *Journal of Computer and System sciences* 10.3 (1975), pp. 384–393.

[8] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *J. ACM* 20.1 (Jan. 1973), pp. 46–61. ISSN: 0004-5411. DOI: 10.1145/321738.321743. URL: https://doi.org/10.1145/321738.321743.

[9] Sudarshan K. Dhall and C. L. Liu. "On a Real-Time Scheduling Problem". In: *Operations Research* 26.1 (1978), pp. 127–140. ISSN: 0030364X, 15265463. URL: http://www.jstor.org/stable/169896 (visited on 05/24/2022).

[10] Dong-Ik Oh and T. P. Bakker. "Utilization Bounds for N-Processor Rate MonotoneScheduling with Static Processor Assignment". In: *Real-Time Syst.* 15.2 (Sept. 1998), pp. 183–192. ISSN: 0922-6443. DOI: 10.1023/A:1008098013753. URL: https://doi.org/10.1023/A:1008098013753.

[11] J. M. López, J. L. Díaz, and D. F. García. "Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems". In: *Real-Time Systems* 28

(Oct. 2004), pp. 39–68. ISSN: 1573-1383. DOI: 10.1023/B:TIME.0000033378.56741.14. URL: https://doi.org/10.1023/B:TIME.0000033378.56741.14.

[12] Sanjoy Baruah. “Partitioned EDF scheduling: a closer look”. In: *Real-Time Systems* 49 (Nov. 2013), pp. 715–729. ISSN: 1573-1383. DOI: 10.1007/s11241-013-9186-0. URL: https://doi.org/10.1007/s11241-013-9186-0.

[13] R. L. Graham. “Bounds for certain multiprocessing anomalies”. In: *The Bell System Technical Journal* 45.9 (1966), pp. 1563–1581. DOI: 10.1002/j.1538-7305.1966.tb01709.x.

[14] R. L. Graham. “Bounds on Multiprocessing Timing Anomalies”. In: *SIAM Journal on Applied Mathematics* 17.2 (1969), pp. 416–429. DOI: 10.1137/0117039. eprint: https://doi.org/10.1137/0117039. URL: https://doi.org/10.1137/0117039.

[15] Alessandra Melani et al. “Response-Time Analysis of Conditional DAG Tasks in Multiprocessor Systems”. In: *2015 27th Euromicro Conference on Real-Time Systems*. 2015, pp. 211–221. DOI: 10.1109/ECRTS.2015.26.

[16] Robert D. Blumofe and Charles E. Leiserson. “Scheduling Multithreaded Computations by Work Stealing”. In: *J. ACM* 46.5 (Sept. 1999), pp. 720–748. ISSN: 0004-5411. DOI: 10.1145/324133.324234. URL: https://doi.org/10.1145/324133.324234.

[17] Sanjoy Baruah. “Federated Scheduling of Sporadic DAG Task Systems”. In: *2015 IEEE International Parallel and Distributed Processing Symposium*. 2015, pp. 179–186. DOI: 10.1109/IPDPS.2015.33.

[18] Son Dinh, Christopher Gill, and Kunal Agrawal. “Efficient Deterministic Federated Scheduling for Parallel Real-Time Tasks”. In: *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2020, pp. 1–10. DOI: 10.1109/RTCSA50079.2020.9203660.

[19] Meiling Han et al. “Federated scheduling for Typed DAG tasks scheduling analysis on heterogeneous multi-cores”. In: *Journal of Systems Architecture* 112 (2021), p. 101870. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2020.101870. URL: https://www.sciencedirect.com/science/article/pii/S1383762120301521.

[20] E. Bini and G.C. Buttazzo. “Biasing effects in schedulability measures”. In: *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004*. 2004, pp. 196–203. DOI: 10.1109/EMRTS.2004.1311021.

A

Appendix 1

A.1 Social-Aware Partitioning

Here we provide the definition of social-awareness for the partitioning problem which is equivalent to the traditional bin-packing problem. First, we provide the corresponding Augmentation Cost Function. This ACF is then used to define social-awareness for partitioning.

Recall, the goal of ACF is to measure the cost of a certain augmentation. If the problem is to partition a set of items on to a set of bins the augmentation under consideration is an addition to the assigned set. In other words, we measure the cost of adding a set of items for each bin. Since the objective of partitioning is that no bin is assigned too many items, we want to give the item to the bin that receives the smallest cost. Recall, social-awareness considers potential sets which includes all of the yet unassigned items. This is the case also for partitioning since when we measure the cost we want to know how big is the cost of a particular item in relation to all the other remaining items. This means that the bins will be assumed to be overfull also for partitioning.

Definition 4. *The Augmentation Cost Function (ACF) for partitioning defines the cost of a certain augmentation of the items assigned to bin b_x from I_a to I_b where $I_a \subset I_b$*

$$ACF_{part}(b_x, I_a, I_b) = \frac{S_x(I_b) - S_x(I_a)}{S_x(I_b)} \cdot \frac{S_x(I_b) - L_x}{L_x} \tag{A.1}$$

$$S_x(I_b) > L_x \tag{A.2}$$

The ACF in equation A.1 measures the cost of adding the augmentation set. As in the case of clustering I_a are the assigned items before the augmentation and I_b are the assigned items after the augmentation. Since the augmentation is now an addition the augmentation set becomes $I_b \setminus I_a$. This gives that the degree to which the augmentation set contributes to the total size is the size of the augmentation set divided by the size of I_b. Thus, the contribution is $\frac{S_x(I_b)-S_x(I_a)}{S_x(I_b)}$. Like in the case of clustering, the relative augmentation distance gives the distance between the limit and the total item size after the augmentation (I_b). However, this is now a measurement of how far the bin is from its objective (i.e. reaching to or below the limit). This means that the cost of an augmentation is increased when the relative augmentation distance increase. Therefore $\frac{S_x(I_b)-L_x}{L_x}$ is multiplied by the contribution to get the final ACF. Like in the case of clustering the ACF is defined

only when $S_x(I_b) \geq L_x$, since when $S_x(I_b) < L_x$ the ACF will be negative for any positive contribution. Then a higher contribution gives a smaller cost, which does not provide meaningful values. However, note that when $S_x(I_b) \leq L_x$ bin b_x can fit all the items in I_b. Since I_b contains all the remaining items, giving I_b to b_x solves the partitioning problem. This ACF is used to define a social-aware partitioning algorithm in definition 5.

Definition 5. *Consider a partitioning algorithm A that assigns a set of items I to a set of bins B. For each iteration k an item $i_x \in I$ is assigned to some bin in $B^k = \{b_j | b_j \in B \wedge S_j(I_j^k \cup \{i_x\}) \leq L_j\}$ where I_j^k is the items already assigned to b_j in iterations leading up to k. The augmentation cost $\phi_x^{j,k}$ of including item i_x in b_j's potential set $I_j'^k$ is defined as $ACF_{part}(b_j, I_j'^k \setminus \{i_x\}, I_j'^k)$. A is **social-aware** iff during no iteration k it assigns item i_x to a bin $b_j \in B^k$ where $\phi_x^{j,k}$ is defined, if there exist some other bin $b_i \in B^k \wedge b_i \neq b_j$ with an undefined $\phi_x^{i,k}$, or with a defined $\phi_x^{i,k}$ such that $\phi_x^{j,k} > \phi_x^{i,k}$.*

Definition 5 states that a social-aware partitioning heuristic gives each item one-by-one to some bin that can fit it (B^k). If $B^k = \emptyset$ for any iteration k then the algorithm fails since there is no bin that can fit the item. This definition use ACF together with potential sets to define the cost $\phi_x^{j,k}$ of including i_x in the potential set $I_j'^k$ of b_j. Note, this is the same as adding the augmentation set to a potential set that does not already include it. In contrast to definition 3 each item must go to the bin with the smallest cost. However, Just like in definition 3 if there exist one or more bins with an undefined cost the algorithm must assign the item to one of those bins. Which of those bins is assigned the item is not defined by social-awareness since any of the alternatives solves the partitioning problem. In conclusion, each item is assigned to bin that is impaired by it least.

A.2 Processor Values

Appendix A.2.1 describes the original processor value provided by [2]. Appendix A.2.2 and A.2.3 provides alternative equations for the processor value. These are meant to capture the value of a processor in more detail. Furthermore, all of the provided processor value equations can be made "fair" as explained in appendix A.2.4

A.2.1 Original processor value

Processor value PV_x^j represent how valuable processor μ_x is to task τ_j. In the original processor value from [2], shown in equation A.3, each processor has a speed-level for each subtask (i.e. fastest processor has speed-level one, second fastest processor has speed-leve two etc.). Thus, the processor value of μ_x to τ_j is computed as such. For each processor, with speed-level s, the number of subtasks that has μ_x as its sth speed-level processor is counted in $cnt_{s,x}^j$. This is divided by s to give a lower value

to processors that are at lower speed-levels.

$$PV_x^j = \sum_{s=1}^{P} \frac{1}{s} \cdot cnt_{s,x}^j \tag{A.3}$$

A.2.2 Speed-based processor value

An alternative to the original processor value is shown in equation A.4. Here the processor value is defined as the sum of the speeds $\delta_{j,i}^x$ of processor μ_x for all subtasks in $\tau_{j,i}$ in τ_j. This aims to more accurately reflect the capacity of the processor.

$$PV_x^j = \sum_{\tau_{j,i} \in \tau_j} \delta_{j,i}^x \tag{A.4}$$

Processor value definition where the processor value is defined by the sum of the speeds of processor μ_x over all subtasks.

A.2.3 Workload-based processor value

This version of processor value, found in equation A.5, introduce weights to each subtask and each task. A subtask that has on average a long execution time across all processors has a greater need for a fast processor. For instance, consider there are two subtasks $\tau_{j,1}$ and $\tau_j 2$ with execution times $C_{j,1} = 2$ and $C_{j,2} = 10$. If processor μ_x provides a double speed to both subtasks their WCET is cut in half such that $C_{j,1} = 1$ and $C_{j,2} = 5$. This means that $C_{j,2}$ has decreased more than $C_{j,1}$ in absolute execution time. Thus, μ_x provides greater value to $\tau_{j,2}$ than $C_{j,1} = 2$ and $C_{j,1}$. Therefore, subtasks that, on average takes longer to execute gets a higher weight. Furthermore, tasks with a short deadline need more processing power to finish the same workload under a shorter time duration. Therefore, the task is given a weight that is the inverse of the deadline.

$$PV_x^j = \frac{1}{D_j} \sum_{\tau_{j,i} \in \tau_j} \delta_{j,i}^x \cdot C_{j,i}^{average} \tag{A.5}$$

A.2.4 Fair processor value

A processor value can be made "fair" through a division by the *processor worth*, i.e. the sum of all processor values over all tasks. This gives a value of the processor that is relative how all the other tasks value that processor. Thus, the task with the highest relative processor value should be the task that "needs" it the most.

$$\beta_{x,}^j = \frac{PV_x^j}{PW_x} \tag{A.6}$$

B

Appendix 2

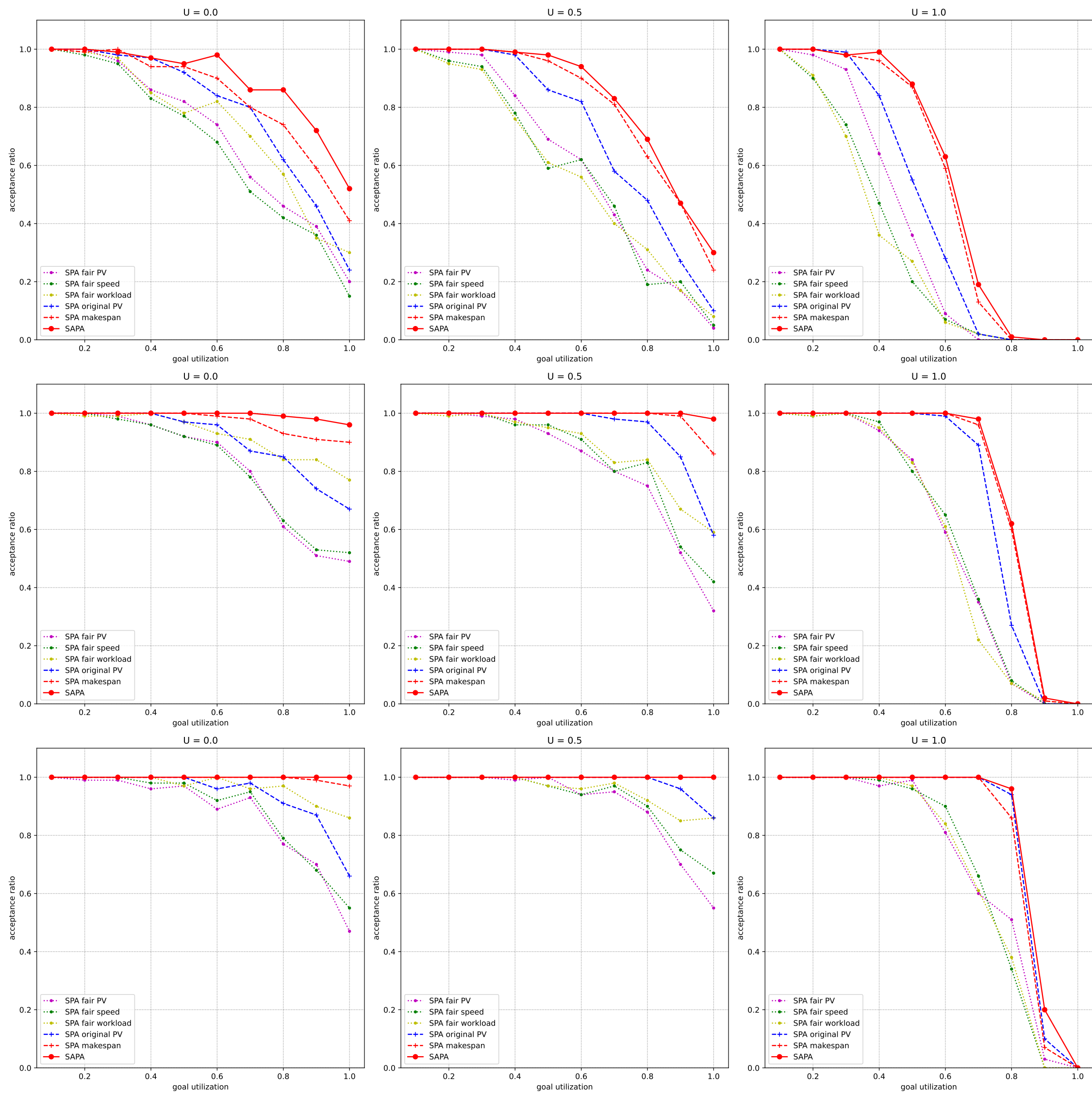


Figure B.1: Acceptance-ratio on a heterogeneous platform of 10 processors with varying degree of uniformity (columns) and increasing deadline multiplier (rows)

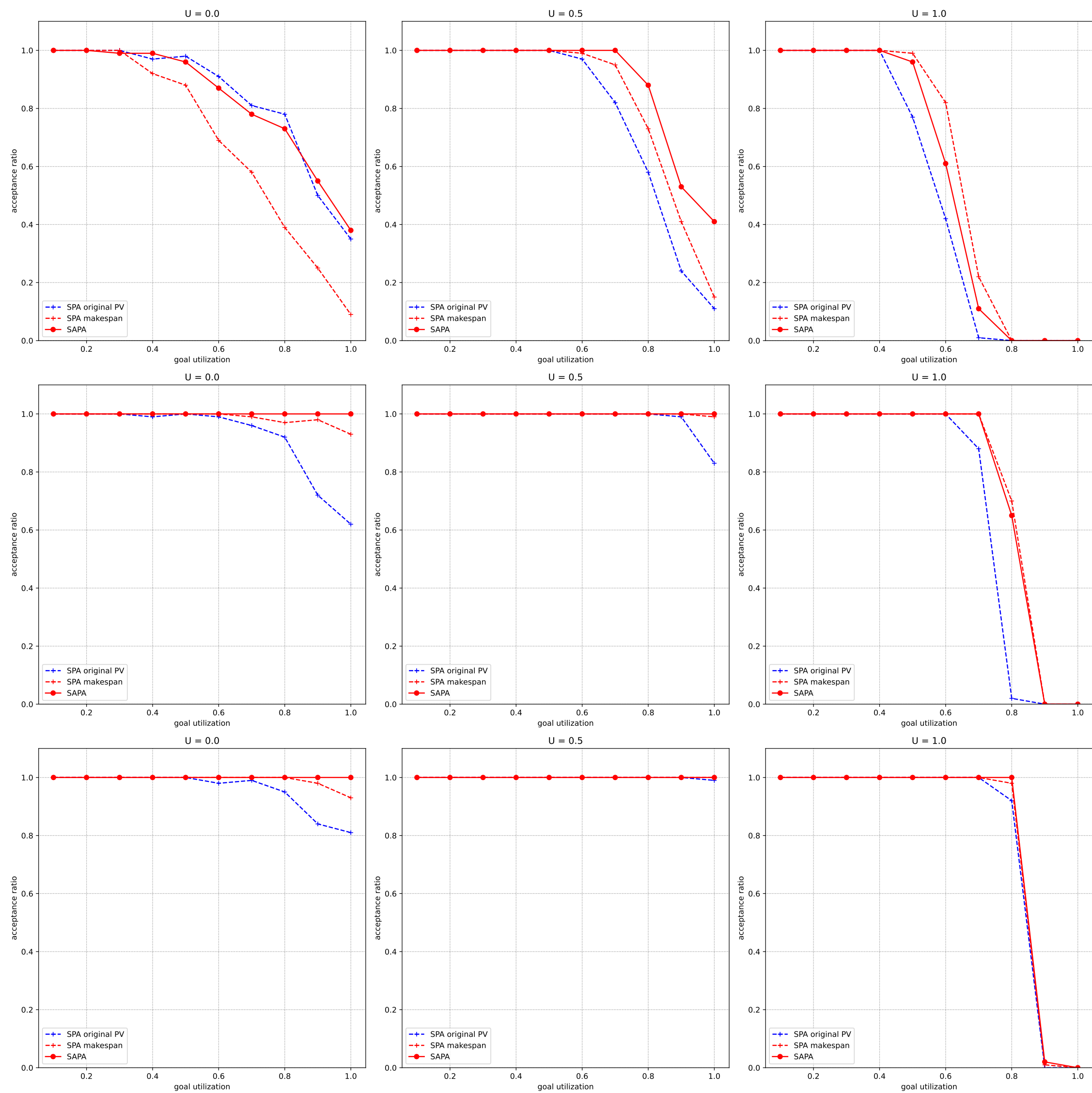


Figure B.2: Acceptance-ratio on a heterogeneous platform of 32 processors with varying degree of uniformity (columns) and increasing deadline multiplier (rows)

Figure B.3: Acceptance-ratio on a homogeneous platform of 10 processors with varying degree of uniformity (columns) and increasing deadline multiplier (rows)

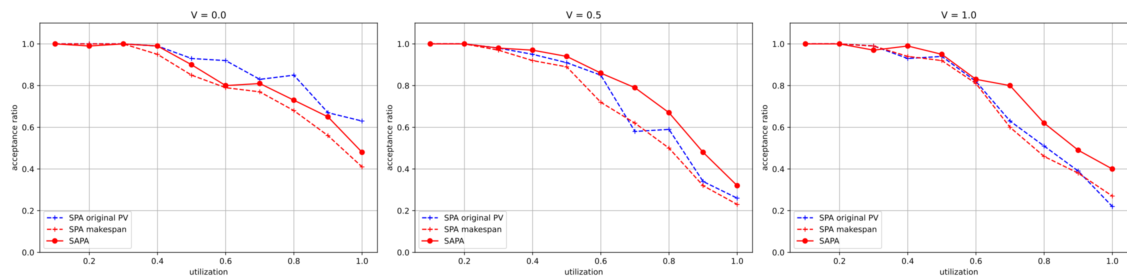


Figure B.4: Acceptance-ratio on a highly unrelated platform of 10 processors with varying degree of subtask variance (columns)

C

Appendix 3

The Integer_UUnifast is a modified UUnifast algorithm presented in [20], which given a number of elements (elem) and a number, n, returns an array of size elem with the sum of all elements equal n and each element is an integer greater than or equal to 1.

Algorithm 4: Integer_UUnifast

---

input : $n \in \mathbb{N}$, $elem \in \mathbb{N}$

output: vectN[elem]

1 $sumU \leftarrow n - elem$;
2 $rest \leftarrow 0$;
3 **for** *i=1:elem-1* **do**
4 $\quad nextSumU \leftarrow (sumU + rest) * \mathbf{rand}(0,1)^{1/(elem-i)}$;
5 $\quad rest \leftarrow \mathbf{round}(sumU - nextSumU) - (sumU - nextSumU)$;
6 $\quad nextSumU \leftarrow \mathbf{round}(nextSumU)$;
7 $\quad vectN[i] \leftarrow 1 + sumU - nextSumU$;
8 $\quad sumU \leftarrow nextsumU$;
9 **end**
10 $vectN[0] \leftarrow 1 + sumU$;
11 **return** *vectN*;