THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Scheduling techniques to improve the worst-case execution time of real-time parallel applications on heterogeneous platforms

PETROS VOUDOURIS

Division of Computer Engineering
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2021

Scheduling techniques to improve the worst-case execution time of real-time parallel applications on heterogeneous platforms

PETROS VOUDOURIS

Technical Report No 196D
Department of Computer Science & Engineering
Division of Computer Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Digitaltryck,
Gothenburg, Sweden 2021.

Abstract

The key to providing high performance and energy-efficient execution for hard real-time applications is the time predictable and efficient usage of heterogeneous multiprocessors. However, schedulability analysis of parallel applications executed on unrelated heterogeneous multiprocessors is challenging and has not been investigated adequately by earlier works.

The unrelated model is suitable to represent many of the multiprocessor platforms available today because a task (i.e., sequential code) may exhibit a different work-case-execution-time (WCET) on each type of processor on an unrelated heterogeneous multiprocessors platform. A parallel application can be realistically modeled as a directed acyclic graph (DAG), where the nodes are sequential tasks and the edges are dependencies among the tasks. This thesis considers a sporadic DAG model which is used broadly to analyze and verify the real-time requirements of parallel applications. A global work-conserving scheduler can efficiently utilize an unrelated platform by executing the tasks of a DAG on different processor types. However, it is challenging to compute an upper bound on the worst-case schedule length of the DAG, called makespan, which is used to verify whether the deadline of a DAG is met or not. There are two main challenges. First, because of the heterogeneity of the processors, the WCET for each task of the DAG depends on which processor the task is executing on during actual runtime. Second, timing anomalies are the main obstacle to compute the makespan even for the simpler case when all the processors are of the same type, i.e., homogeneous multiprocessors. To that end, this thesis addresses the following problem: How we can schedule multiple sporadic DAGs on unrelated multiprocessors such that all the DAGs meet their deadlines.

Initially, the thesis focuses on homogeneous multiprocessors that is a special case of unrelated multiprocessors to understand and tackle the main challenge of timing anomalies. A novel timing-anomaly-free scheduler is proposed which can be used to compute the makespan of a DAG just by simulating the execution of the tasks based on this proposed scheduler. A set of representative task-based parallel OpenMP applications from the BOTS benchmark suite are modeled as DAGs to investigate the timing behavior of real-world applications. A simulation framework is developed to evaluate the proposed method. Furthermore, the thesis targets unrelated multiprocessors and proposes a global scheduler to execute the tasks of a single DAG to an unrelated multiprocessors platform. Based on the proposed scheduler, methods to compute the makespan of a single DAG are introduced. A set of representative parallel applications from the BOTS benchmark suite are modeled as DAGs that execute on unrelated multiprocessors. Furthermore, synthetic DAGs are generated to examine additional structures of parallel applications and various platform capabilities. A simulation framework that simulates the execution of the tasks of a DAG on an unrelated multiprocessor platform is introduced to assess the effectiveness of the proposed makespan computations. Finally, based on the makespan computation of a single DAG this thesis presents the design and schedulability analysis of global and federated scheduling of sporadic DAGs that execute on unrelated multiprocessors.

Keywords

Hard real-time systems, parallel applications, heterogeneous multiprocessors, unrelated model, global, federated, work-conserving, DAG, makespan, response time

Acknowledgment

First of all, I would like to express my sincere gratitude to my advisors Per Stenström and Risat Pathan, for giving me the opportunity to pursue a Ph.D. degree under their supervision. Without their longstanding guidance, support and patient, this thesis would not be possible.

I would like to thank my examiner Fredrik Dahlgren for his insightful feedback. Also, I would like to thank Vassilis Papaefstathiou, Miquel Pericas, Madhavan Mannivanan, Pedro Petersen Moura Trancoso, Yiannis Sourdis, and Jan Jonsson for their valuable feedback and guidance all these years. I would like to thank my colleagues and friends at Chalmers for the joyful working environment and the after-work activities. Last but not least, I would like thank to my family and my friends for their continuous support during my studies

This research was funded by the MECCA project under the ERC grant ERC-2013-AdG 340328-MECCA.

List of Publications

Appended publications

This thesis is based on the following publications:

I Petros Voudouris, Per Stenström, Risat Pathan "Timing-Anomaly Free Dynamic Scheduling of Task-Based Parallel Applications"
IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017.

II Petros Voudouris, Per Stenström, Risat Pathan "Bounding the Execution Time of Parallel Applications on Unrelated Multiprocessors"
Real-Time Systems journal, Springer (Under revision), 2021.

III Petros Voudouris, Per Stenström, Risat Pathan "Response time analysis for globally scheduled sporadic DAGs on unrelated multiprocessors"
Manuscript, 2021.

IV Petros Voudouris, Per Stenström, Risat Pathan "Federated scheduling of sporadic DAGs on unrelated multiprocessors"
Under submission, 2021.

Other publications

The following publications were published during my PhD studies, but they are not part of this thesis.

V Risat Pathan, Petros Voudouris, Per Stenström "Scheduling Parallel Real-Time Recurrent Tasks on Multicore Platforms"
IEEE Transactions on Parallel and Distributed Systems (TPDS), 2018.

VI Petros Voudouris, Per Stenström, Risat Pathan "Bounding the Execution Time of Task-based Parallel Applications on Unrelated Multiprocessors"
Technical report, https://research.chalmers.se/en/publication/505944, 2018.

VII Petros Voudouris, Per Stenström, Risat Pathan "A Safe and Tight Estimation of the Worst-Case Execution Time of Dynamically Scheduled Parallel Applications"
MULTIPROG workshop, High-Performance and Embedded Architectures and Compilers (HiPEAC), 2016.

VIII Petros Voudouris, Per Stenström, Risat Pathan "Timing-Anomaly Free Dynamic Scheduling of Task-Based Parallel Applications"
IEEE Real-Time Systems Symposium (RTSS), Work in progress, 2016.

Contents

Chapter 1

Introduction

1.1 Background

More and more everyday life devices are getting digitized by incorporating some computing system to enhance the device's functionalities. The computing system understands the device's environment by using sensors and reacts to it with some mechanical actuators. For example, a car's braking assistant system based on the sensor inputs adapts how it is working to improve the safety of the passengers and the pedestrians around it. As more functionalities are automated, the software that controls the device becomes more complex, and it needs to compute more data. For example, avionics software size has increased significantly over the last years [1, 2]. Also, more advanced sensors based on radar and vision technologies [3, 4] are used to get a better understanding of the environment but generate a large amount of data that the computing system needs to process.

A *hard real-time system* is a computing system embedded in some device that needs to interact with its environment within strict time constraints. Hard real-time systems can be found, for example, in space applications, avionics, automotive, telecommunication, industrial manufacture technologies, and medical equipment [5] and they represent a significant share of the semiconductor industry revenue [2]. More formally, a hard real-time system, illustrated in Figure 1.1, is a system which is composed of (i) a software application that is embedded on a known hardware platform — embedded system and (ii) by a quantitative timing analysis for the embedded system that rigorously proves that the application completes its execution before a given deadline.

Heterogeneous multiprocessor platforms can provide the necessary performance and energy gains for parallel applications [7–14]. To be able to use heterogeneous multiprocessors for hard real-time systems, however, we need to ensure *time predictability*; one has to guarantee the timeliness of a real-time parallel application that is executed on a heterogeneous platform by designing an effective scheduling algorithm and doing the offline schedulability analysis.

Related work on scheduling real-time parallel applications for heterogeneous multiprocessors has focused on restricted application models [15–20] or platform [21–27] which limit their applicability to practical problems. A parallel application that is modeled as a directed acyclic graph (DAG), where the nodes represent sequential tasks and edges represent dependencies among the tasks, is suitable to model many real-time control and monitoring applications because it represents the parallelism and

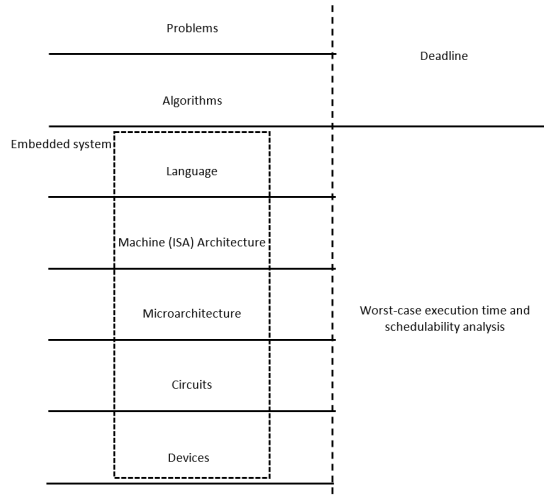


Figure 1.1: The left part of the figure are the levels of transformation for an application and with the dashed rectangle we present the embedded system. The right part of the figure presents the timing analysis needed to check the timing requirements of a hard real-time system. The deadline is determined by the problem and the algorithm. The WCET and schedulability analysis verify that the hard real-time system can meet its deadline. The figure is based on Fig. 1.6, in [6].

its limitations (e.g., data dependencies) that exist in the application. The *unrelated* heterogeneous multiprocessor model associates the worst-case execution time (WCET) of each task with each processor type, and it can model many of the available heterogeneous multiprocessors. Under a global (dynamic) work-conserving scheduler for the tasks of a DAG, a processor is never idle if there are available tasks to execute. Then, it is possible to efficiently utilize unrelated multiprocessors by avoiding processor load imbalance issues that partitioned (static) scheduling approaches suffer from. However, the timing analysis is challenging.

First, during runtime, a task can execute on any processor type, and it is not clear offline what WCET we need to consider for each task in order to determine the worst-case schedule length of a DAG, called *makespan*. Second, even if we assume that there is one processor type in the multiprocessor platform which means that it is a *homogeneous* platform, the execution time of a globally scheduled parallel application may increase when some tasks take less time than their WCETs at runtime. This is known as an *execution-time-based timing anomaly* [28–30], which comprises one of the main obstacles to minimize the pessimism for the calculation of the makespan when we consider *any* work-conserving scheduler. Previous work on homogeneous multiprocessors [31] that compute the makespan introduce a closed-form, analytical formula which in order to avoid timing anomalies assumes that no task can run in parallel with the tasks that belong to the critical path of the DAG. This is pessimistic and leads to an overestimation of the makespan. Finally, no previous work computes the makespan of a globally scheduled DAG on unrelated multiprocessors, to the best of my knowledge. Without knowing how to compute the makespan of a single DAG, we cannot analyze a hard real-time system that is modeled using the broadly adopted sporadic DAGs model [32, 33] for unrelated multiprocessors. This limits the use of

heterogeneous multiprocessors to hard real-time systems.

1.2 Problem statement

This thesis first investigates the problem of how we can improve the makespan computation of a globally scheduled DAG that executes on a homogeneous multiprocessor such that we reduce the pessimism of the estimation that is needed to avoid timing anomalies. This problem is addressed in **Paper I**. Next, we approach the problem of how we can schedule multiple DAGs on an unrelated multiprocessor platform such that we can guarantee the schedulability of the DAGs, i.e., all the DAGs meet their deadlines. First, **Paper II** finds an approach to compute the makespan of a single DAG on an unrelated heterogeneous platform. Next, we need to find a way to schedule multiple DAGs that are sharing the unrelated platform by upper bounding the longest time that a DAG can be delayed by other DAGs, such that we can check if all the DAGs meet their deadlines. To address the last problem, two approaches are proposed in **Paper III** and **Paper IV**.

1.3 Contributions

This thesis, that is based on four papers, makes the following contributions:

- **Paper I**: This paper introduces a scheduler, called `Lazy` , to execute the tasks of a DAG on a homogeneous platform that is proven to be execution-time-based timing-anomaly free. Based on the time predictable execution of the tasks, a novel approach that simulates the execution of a DAG to compute the makespan is introduced that provides a tighter and more scalable estimation of the makespan, with respect to the number of processors, in comparison with the state-of-the-art in [31].

- **Paper II**: This paper presents a scheduler, called $\mathcal{GHE}$, to execute the tasks of a single DAG on unrelated multiprocessors. The scheduler has proven to possess a property called the *greediness property* that allows us to find the minimum processing capability and the maximum processing wastage for an unrelated platform that executes a parallel application modeled as a DAG. We use the greediness property of the scheduler to introduce two approaches that make a trade-off between computational complexity and tightness when computing a safe upper bound on the makespan of a DAG that executes on unrelated multiprocessors. The homogeneous and related (uniform) multiprocessors are special cases of the unrelated multiprocessor model, and the proposed methods specialize in the makespan computations presented in the state-of-the-art [28] and [23]. Finally, the makespan computation applies to any static- or dynamic-priority-based work-conserving scheduler that has the greediness property.

- **Paper III**: This paper introduces a global scheduler for *multiple* DAGs that are executed on an unrelated multiprocessor platform. Similar to **Paper II** the scheduler has the greediness property. Based on the greediness property, we introduced the concept of inflated workload to incorporate the quality of heterogeneity of a multiprocessor platform in our proposed schedulability analysis.

We use the inflated workload to compute the response time of globally scheduled sporadic DAGs that are executed on unrelated multiprocessors.

- **Paper IV**: This paper introduces the concept of processor value for a DAG with respect to meeting its deadline on an unrelated heterogeneous platform. The paper uses the scheduler of **Paper II** and introduces a simple and elegant method to compute the makespan of a DAG that executes on unrelated multiprocessors. The processor value and the makespan of a single DAG are used to develop a federated scheduling algorithm that statically allocates a DAG to a dedicated subset of unrelated processors such that all the sporadic DAGs meet their deadlines. The advantage of federated scheduling is that, by assigning dedicated processors to a DAG, there is no interference from the other DAG. However, the challenge is to find the right subset of dedicated processors for each DAG.

1.4 Thesis organization

Chapter 2 presents the summary of the papers included in the thesis, and Chapter 3 concludes the thesis and discusses future research directions.

Chapter 2

Summary of papers

2.1 Paper I

Title: " *Timing-Anomaly Free Dynamic Scheduling of Task-Based Parallel Applications*"

2.1.1 Background

Homogeneous multicore architectures can provide a high time-predictable performance through parallel processing [34–39] of parallel applications [40, 41] that are modeled as DAGs. However, any global (dynamic) work-conserving scheduler can suffer from execution-time-based timing-anomalies [28–30]. The state-of-the-art method to compute the makespan of a DAG that executes on homogeneous multiprocessors [28,31,42] is a closed-form, analytical approach that is applicable for work-conserving scheduling under *any* priority-ordering of the tasks, for example, earliest-deadline-first (`EDF`), rate monotonic (`RM`), depth-first search (`DFS`) and breadth-first search (`BFS`). The state-of-the-art, closed-form formula to compute an upper bound on the makespan of a DAG that executes on a homogeneous platform [28, 31, 42] is presented in Eq. (2.1).

$$T_M \leq \mathcal{W}_\infty + \frac{(\mathcal{W}_1 - \mathcal{W}_\infty)}{M} \tag{2.1}$$

where:

- M: The number of processors of a homogeneous multiprocessor platform.
- $\mathcal{W}_1$: The total workload of a parallel application is modeled as a directed acyclic graph (DAG). The total workload is computed by summing the WCET of all the tasks.
- $\mathcal{W}_\infty$: The workload of the tasks that belongs to the critical path of the application. The critical path is computed by summing the WCET of all the tasks that belong to the longest source-to-sink path of the DAG (also called the critical path in this thesis).

While Eq. (2.1) is easy to compute, the main disadvantage is the assumption that the tasks that belong to the critical path are interfered by all the other tasks. This assumption is pessimistic because, at runtime, there are tasks that can be executed in parallel with the critical path's tasks.

2.1.2 Problem statement

How can we improve the makespan computation of a single globally scheduled parallel application that is modeled as a DAG and executes on a homogeneous multiprocessors platform?

2.1.3 Contributions

This paper contributes with the design of a timing-anomaly-free global scheduling method, called `Lazy`, which is non-preemptive and non-work-conserving, in the sense that some ready tasks may not be dispatched for execution even if some processors are idle. Unlike all other earlier approaches where the runtime simulation of the tasks' execution cannot be used to find the makespan due to timing anomalies, the proposed `Lazy` scheduler can determine the makespan by simply executing the tasks of the DAG while avoiding any timing anomaly. In other words, the simulation of the schedule of the DAG is proved to provide a safe upper bound on the makespan.

The main design idea of `Lazy` is that each task in the DAG is assigned a fixed priority. The priority of a child task is assigned based on the priority of its parent with a constant time computation. So, when a task (parent) is currently in execution, we can compute in constant time the priorities of the tasks (children) that *will* spawn in the future when the parent finishes its execution. The dispatch condition checks if all the highest priority tasks have already been dispatched, and if not, it checks whether there are *available processors for all the higher priority tasks* that may come in the future. If it is true, a task is dispatched to an idle processor. Otherwise, a task is not dispatched for execution even if some processor is idle, which means that `Lazy` is non-work-conserving, and this property is crucial to avoid timing anomalies.

The `Lazy` scheduler is able to dispatch a lower-priority task — *out of strict decreasing-priority order* — if there are enough processors to execute the higher priority tasks that are ready for execution or may become ready in the future. Consequently, it is guaranteed that a lower priority task during actual runtime cannot start its execution later compared to the starting time that is used offline for the estimation of the makespan.

To prove anomaly freedom, we compare two schedules of the DAG on the same platform. We compare the schedule S_{WCET} where all tasks execute exactly for their WCET and schedule S where some tasks may execute for less than their WCET. Note that S_{WCET} is the schedule we use offline to compute the makespan while schedule S mimics any arbitrary schedule at runtime. The starting time of any task in S can be as late as the starting time in S_{WCET} since `Lazy` has already dispatched or has reserved processors for all its higher priority tasks. We have assumed that the scheduler is non-preemptive, and consequently, the same holds for their completion time. So the makespan of S cannot be longer than S_{WCET}, which shows that Lazy is a timing-anomaly free scheduler.

2.1.4 Summary of results

To assess the effectiveness of the `Lazy` scheduler in determining the makespan of parallel applications, we study its performance in the dynamic scheduling of Fibonacci, Sort, Strassen, and FFT task-based parallel OpenMP applications from the BOTS benchmark suite [40]. As a baseline, we use the state-of-the-art given by Eq. (2.1). We use *tightness* and *scalability* (based on Gustafson's Law [43]) as key metrics to

compare the effectiveness of `Lazy` in determining the makespan with that of using the baseline.

For all the cases, the estimation of the makespan of the `Lazy` scheduler is tighter than the state-of-the-art for each application. The worst-case assumption in Eq. (2.1) is that the tasks that are not on the critical path do not run in parallel (i.e., always interfere) with the tasks of the critical path. However, the structure of a DAG may allow the tasks in the critical path to execute in parallel with tasks that are not part of the critical path. For different applications and a different number of processors, the simulation of the `Lazy` scheduler achieves on average 9% and a maximum of 36% tighter estimation of the makespan in comparison to the state-of-the-art. Furthermore, for all the applications, the `Lazy` scheduler scales better or similar to the baseline. For the different applications and configurations, the increase in scalability of the `Lazy` scheduler in comparison to the baseline is on average 14% and maximally 30%.

2.2 Paper II

Title: " *Bounding the Execution Time of Parallel Applications on Unrelated Multiprocessors*"

2.2.1 Background

Heterogeneous multiprocessors can offer high performance at low energy expenditures [7–14]. However, to be able to use them in hard real-time systems, timing guarantees need to be provided [33]. One of the main challenges is to determine the worst-case schedule length (also known as makespan) of a parallel application that executes on unrelated multiprocessors. The unrelated multiprocessor is a generalization of homogeneous multiprocessors, so the problem of timing anomalies still holds for a globally scheduled DAG on unrelated multiprocessors.

The work in [15–20] consider the scheduling of independent and sequential tasks on an unrelated platform. Unfortunately, sequential independent tasks cannot model the data or other dependencies among the tasks that exist in parallel applications. Other work [21–27] on scheduling directed acyclic graphs (DAGs) on unrelated platform assume that a node can only execute on (i.e., is compatible with) exactly one type of processor. Limiting the compatibility of the tasks through such modeling reduces the actual parallelism that we can achieve for a parallel application on heterogeneous multiprocessors [12, 13] which in turn increases the execution time of an application. The work in [23] uses global (dynamic), work-conserving scheduling for the tasks of a DAG. The tasks of the DAG can execute on any processor and can efficiently utilize a heterogeneous platform. However, [23] uses the related (uniform) multiprocessor model that associates each processor with a speed factor, meaning that *all* tasks of a parallel application can benefit equally by a processor type, which is unrealistic.

The DAG model for a single parallel application in combination with the unrelated multiprocessor platform model can realistically represent many of today's parallel applications and heterogeneous platforms. A global scheduler can execute the tasks of a DAG at any processor type of a platform by efficiently utilizing a heterogeneous platform and may complete the execution of the application relatively earlier. However, no previous work considers determining the makespan of a parallel application on an unrelated multiprocessor platform.

2.2.2 Problem statement

How can we compute the worst-case schedule length, i.e., makespan, of a globally scheduled DAG that is executed on unrelated multiprocessors?

2.2.3 Contributions

This paper introduces a global work-conserving scheduler, called $\mathcal{GHE}$, that is used to schedule the tasks of a DAG on an unrelated multiprocessor platform. The $\mathcal{GHE}$ scheduler, at each scheduling decision point, first checks if the tasks that are already in execution can *migrate* to some faster processor. Next, if there are tasks in the ready queue and there are idle processors, the $\mathcal{GHE}$ scheduler starts *dispatching* one-by-one new task awaiting execution in the ready queue on the fastest idle processor.

Because of the dynamic nature of the scheduler, we do not know at which processor(s) a task may be executed (dispatched or migrated), so it is not clear what WCET we need to consider to find the makespan of the DAG since the same task may have different WCET on different processor type on an unrelated multiprocessor platform. To model the total workload of the application and the workload of the critical path, we use the *minimum* WCET of each task of the DAG to succinctly capture the workload using only two parameters. Then we find how much a task can be delayed because during runtime it may execute on a processor that is slower compared to the processor that provides the minimum WCET.

The $\mathcal{GHE}$ scheduler is proven to have a property called the *greediness property*, which enables us to identify the worst-case task-processor mapping that may happen during run time by considering both the workload and the heterogeneity of the platform during the schedulability analysis. We define the minimum processing capability and the maximum processing idleness of the unrelated platform concerning the DAG under analysis based on the greediness property. One important characteristic of the $\mathcal{GHE}$ scheduler is that the greediness property is oblivious to the task priority order, so it holds for any static- or dynamic-priority assignment of the tasks, for example `EDF`, `RM`, `DFS`, and, `BFS`.

By combining the workloads of the application with the minimum processing capacity and the maximum processing idleness that we can have because of the dependencies of the DAG or due to lack of parallelism, we find the makespan with two approaches. First, an exhaustive search-based approach, called `Comb`, is proposed to find the makespan. `Comb` has a high computational complexity in order to find a tighter makespan and it is suitable for small applications that have tight deadlines. To reduce the computational complexity of the `Comb`, a polynomial-time approach, denoted by `Fast`, is introduced that can be used to find a relatively less tight makespan which is well suited not only for small applications but also for a large applications with relatively less tight deadlines.

2.2.4 Summary of results

To quantitatively evaluate the proposed makespan computation, we use Fibonacci, Sort, Strassen, and FFT task-based parallel OpenMP applications from the BOTS benchmark suite [40] suite and synthetic DAGs. Because there is no related work that considers the same general system model as ours, we compare the `Fast` to the exhaustive approach `Comb` to identify how much tightness we need to sacrifice in order to avoid the high time complexity of `Comb` and to compute the makespan

in polynomial time with `Fast`. Next, we compare `Fast` to an approach based on simulation setup, denoted by Sim, to find the pessimism of `Fast` compared to the schedule length obtained under the simulation setup. Sim is simply execution of the tasks of a DAG by forcing each task to execute for its WCET under the $\mathcal{GHE}$ scheduler. The schedule length in Sim is a lower bound on the makespan of the application, which shows the space for potential improvement that we can achieve in case we could compute the makespan optimally.

From the simulation results of the parallel *applications*, we have seen that `Fast` over-approximates the makespan no more than 3% compared to the exhaustive approach `Comb`. Next, for the applications, the `Fast` is on average 19% and, at most, 62% greater than Sim. For the *synthetic* DAGs, we try to explore different DAG structures than the applications and platforms with high heterogeneity. From the results, we observe that `Fast`provides, on average, 6% and at maximum 16%, less tight makespan compared to the `Comb`. Compared to Sim, the `Fast` is, on average, 51% and up to 74% more pessimistic.

We compare this work with related works that make similar assumptions about the application and platform models. In case all the tasks have the same WCET for all the processors (a homogeneous multiprocessor set up), the `Comb` and `Fast` are equal to the makespan as proposed in Eq. (2.1). If the processors have the same speed for all the tasks (a related multiprocessor set up), the two approaches are equal to the state-of-art for related multiprocessors [23]. Finally, by assuming that the tasks of a DAG are compatible with only one processor type, we show that our approaches are equivalent with typed DAGs [24].

2.3 Paper III

Title: " *Response time analysis for globally scheduled sporadic DAGs on unrelated multiprocessors*"

2.3.1 Background

Hard real-time systems usually need to execute multiple applications, for example, to process the input data of several sensors and control different actuators. The sporadic DAG model [32,33] assumes a finite number of recurring DAGs. Each DAG potentially generates an infinite sequence of releases (jobs). The first release can arrive at any time instant, and any two consecutive releases are separated at least by a minimum inter-arrival time (also called the period). Each DAG has a relative deadline that needs to be met, and it is relative to the arrival time of every release of that DAG. The sporadic DAG model has gained attention [22,23,31,44–50] because it is suitable to model many real-time control and monitoring applications. A global scheduler [25,31,33,44,46,49] can execute a task of a DAG on any processor of an unrelated platform. Because the tasks of the DAGs compete for the same processors, they interfere with each other. We need to establish an upper bound on the interference that the DAGs suffer from one another and combine such interference with the makespan to compute overall response time of each DAG task which can be used to test if all DAGs meet their deadlines or not. No previous work has analyzed the global scheduling of sporadic DAGs on unrelated multiprocessors to the best of our knowledge.

2.3.2 Problem statement

How can we globally schedule a set of sporadic DAGs on an unrelated multiprocessor platform such that we can guarantee the schedulability of the DAGs?

2.3.3 Contributions

This paper introduces a scheduler that builds upon the $\mathcal{GHE}$ scheduler from **Paper II** and schedules the tasks of *multiple* sporadic DAGs. Recall that the $\mathcal{GHE}$ scheduler at each scheduling decision point first allows migration of the currently executing tasks to a relatively faster (idle) processor and then dispatches new tasks from the ready queue. The $\mathcal{GHE}$ scheduler from Paper II is extended in this Paper III with preemption capability such that a task of a DAG is allowed to preempt some other lower-priority currently-executing task that belongs to some other DAG. Similar to $\mathcal{GHE}$, this extended scheduler has the greediness property which is oblivious to the priority of the tasks.

One of the main challenges for analyzing the schedulability of multiple DAGs is to account for interference that one DAG may have over another. Computing the exact interference requires one to know the critical instance [51] which is still unknown for global multiprocessor scheduling like the $\mathcal{GHE}$ scheduler. Therefore, we find an upper bound on the interference to compute the response time of each DAG. To establish an upper bound on the interference among the tasks of different processors, we need to take into account the heterogeneity of the processors. This is because a higher-priority task executing on a slow processor interferes with the other lower-priority task for a relatively longer time duration compared to the case when the higher priority task is executing on a relatively faster processor.

To safely upper bound the interference that is introduced by one DAG competing for the same unrelated processors, we increase (i.e., inflate) the workload of a DAG to capture the heterogeneity of the platform. The inflated workload is computed based on the greediness property of the scheduler. The inflated workload encapsulates in a single parameter the worst-case task-processor mapping of the tasks for each DAG of the sporadic DAG set. Finally, we extend the window analysis from the homogeneous platform setup [31, 33, 52, 53] to unrelated multiprocessor platform. The outcome of this extended window-based analysis is to find the total interference that all the higher-priority tasks impose on a lower-priority task under analysis.

Based on the makespan of each DAG and the interference that it can suffer by the other DAGs, we are able to compute the response time of each DAG for *any* scheduler that is work-conserving and possesses the greediness property. In other words, the proposed response time analysis is applicable to a wide range of well-known fixed- and dynamic- priority based schedulers for scheduling a set of sporadic DAGs on an unrelated heterogeneous platform. To that end, we present how our general response time analysis (RTA) is extended for fixed-priority (e.g., RM) and dynamic-priority (e.g., EDF) based schedulers.

2.3.4 Summary of results

An experimental framework is proposed considering multiple sporadic DAGs and an unrelated heterogeneous multiprocessor platform. The simulation results using the experimental framework show that the specialization of the response time analysis improves the deadline acceptance ratio compared to the response time that is oblivious

to the priorities of the DAGs. For example, EDF achieves on average 27% and RM on average 24% higher acceptance ratio in comparison to the scheduler that considers no special priority ordering. Our experiments confirm that as we continue adding slower processors to the experimental framework, the acceptance ratio decreases because some processors are very fast for some tasks while are very slow for others. Overall, the acceptance ratio decreases for $H = 2$, on average by 44%, and for H=4 on average a 99% lower acceptance ratio compared to the case that $H = 1$.

There are at least two sources of pessimism that one has to consider when doing schedulability analysis for multiple DAGs on a heterogeneous platform. First, the variation in the heterogeneous characteristics of the processors for the makespan computation. Second, the analysis of a general work-conserving scheduler also needs to pessimistically consider the worst possible interference that one task suffers from tasks from the other DAGs. The makespan computation of this paper can be improved by using the makespan proposed in **Paper II** or **Paper IV**. The computation of the interference takes advantage of the greediness property of the scheduler and computes with low pessimism the interference among the DAGs.

2.4 Paper IV

Title: " *Federated scheduling of sporadic DAGs on unrelated multiprocessors*"

2.4.1 Background

In this paper, we use the sporadic DAG model [32, 33], as in **Paper II**. In federated scheduling [26, 44, 45, 47, 48, 50] a DAG either gets a dedicated subset of processors (i.e., a cluster) such that it can execute in isolation without being interfered by other DAGs or the DAG executed sequentially upon a single processor as in partitioned scheduling. For homogeneous processors [44, 45, 47, 48, 50] and typed DAGs [26] we need to find the number of processors that each DAG needs in order to meet its deadline. However, for unrelated multiprocessors, we need to find the number of processors and the right *processor type* for each processor that the DAG needs to meet its deadline. No previous work has investigated the federated scheduling of sporadic DAGs on unrelated multiprocessors to the best of our knowledge.

2.4.2 Problem statement

How can we schedule a set of sporadic DAGs on an unrelated multiprocessor platform with federated scheduling in such a way that we can guarantee the schedulability of the DAGs?

2.4.3 Contributions

We use the $\mathcal{GHE}$ scheduler from **Paper II** to schedule the tasks of a single DAG on a dedicated cluster of unrelated multiprocessors, and we introduce an approach to compute the makespan of a single DAG. For the DAGs assigned to a single processor, we use uniprocessor preemptive EDF to schedule sequentially the tasks of the DAGs assigned to this processor. To determine the right cluster for each DAG, such that all the DAGs meet their deadlines, we introduce the notion of *processor value* for each processor that belongs to an unrelated platform. The processor value of a processor is

computed based on the quality of a processor in terms of meeting deadline of each DAG under the $\mathcal{GHE}$ scheduler. A high processor value computed for a DAG means that this processor can execute the DAG tasks relatively faster compared to some other processor with a lower processor value and hence the DAG has higher likelihood of meeting the deadline.

Based on the processor value that the DAGs have for the processors, we develop a federated scheduling algorithm that statically allocates a DAG to a cluster of dedicated unrelated processors. For a DAG that gets a multiprocessor cluster to execute in isolation, based on the analysis of the $\mathcal{GHE}$ scheduler, we compute the makespan to check if it meets its deadline. For the DAGs assigned to a single processor cluster, we use a uniprocessor utilization-based schedulability test to check if all the DAGs meet their deadlines. The federated scheduling algorithm returns *success* if it finds a feasible DAG-cluster allocation for all the sporadic DAGs; otherwise, it returns *failure* to schedule the sporadic DAG set.

2.4.4 Summary of results

To quantitatively evaluate the federated scheduling algorithm, we generate sporadic synthetic DAGs with implicit deadlines. First, we test the schedulability of the proposed federated scheduling by varying the number of processor types. From the simulation results, we have seen that by trading more extended deadlines for cheaper/slower processors (i.e., if someone can afford to have more extended deadlines), one can choose a lower-cost platform and achieve on average a 6% higher acceptance ratio.

Next, we compare our approach to global scheduling for unrelated multiprocessors. For federated scheduling, a DAG with a dedicated cluster cannot interfere with the other DAGs. Furthermore, DAGs scheduled sequentially to a single processor can efficiently utilize the processor because the single processor `EDF` for implicit deadlines is optimal considering meeting the DAGs deadlines. In contrast, for global scheduling, each DAG can minimize its makespan because it can use all the processors but interfere with all the lower priority DAGs. The simulation results show that federated scheduling has on average 80% and 120% higher acceptance ratio than `EDF` and `RM`, respectively.

Finally, we specialize the system model for related multiprocessors to compare our approach to federated scheduling with previous work for related multiprocessors [23]. Even though our approach has a less tight makespan compared to [23] for related multiprocessors, it has on average 87% higher acceptance ratio because our DAG-cluster allocation takes into account the heterogeneity of the processors and assigns suitable processors for the DAGs such that all the DAGs can meet their deadlines. However, the main objective of [23] is to solve the underutilisation of homogeneous multiprocessors. The processor to the DAG allocation mechanism is not aware of the capability of the different processor types. It specifies only the number of processors but does not determine the processor type of each processor a DAG needs to meet its deadline. By assuming that we assign the faster processor to the DAG with the longest makespan, the DAG-cluster assignment becomes identical to our approach. Since the makespan of [23] is tighter, compared to the makespan computation of this paper, by combining the two techniques, a higher acceptance ratio can be achieved.

Chapter 3

Concluding remarks and future work

More and more functionalities of hard real-time systems are automated that require high computational demands. This calls for parallel processing. Heterogeneous multiprocessors provide high performance and high energy efficiency for parallel applications. Global schedulers can efficiently utilize heterogeneous processors. However, the schedulability analysis of parallel real-time applications on heterogeneous multiprocessors is challenging due to the heterogeneity of the processors and the timing anomalies that may arise.

First, we consider a single parallel application that is modeled as a DAG, and we compute the makespan. In **Paper I** we focus on homogeneous multiprocessors, and a novel time predictable scheduler, called `Lazy`, is introduced to schedule the tasks of a DAG. The simulation of the execution of `Lazy` is a safe upper bound on the makespan, in contrast to previous approaches that suffer from timing anomalies and use closed-form, analytical methods. Simulation results show that the proposed approach computes a tighter and more scalable makespan, with respect to the number of processors, than the previous approaches. In **Paper II** we focus on unrelated multiprocessors, and a global, work-conserving scheduler is introduced to schedule the tasks of a DAG. An important characteristic of the scheduler is that it allows tasks to migrate among the heterogeneous processor such that they can enjoy a faster execution. Based on the proposed scheduler, a method to compute the makespan is introduced. Simulation results show that the proposed method computes the makespan tightly compared to an exhaustive approach and with low pessimism compared to the simulation of the execution of the DAG. This is the first attempt to compute the makespan of globally scheduled DAGs that execute on unrelated multiprocessors, to the best of our knowledge.

Second, we consider multiple parallel applications with real-time constraints that are modeled with the sporadic DAG model, and they are sharing an unrelated multiprocessor platform. In **Paper III** a global work-conserving scheduler is introduced to schedule the tasks of multiple DAGs. An upper bound on the interference among the DAGs is computed by taking into account the different processing capabilities of the unrelated processors. Based on the makespan of each DAG and the interference, we compute the response time of each DAG, assuming any priority order of DAGs. Next, we specialize the response time analysis for dynamic (`EDF`) and static (`RM`) priority-

based scheduler. The simulation results show that EDF and RM schedulers achieve a significantly larger acceptance ratio compared to the general approach where DAGs do not have any priorities. In **Paper IV**, a federated scheduling algorithm is introduced that statically assigns dedicated clusters to DAGs to avoid the interference among the DAGs that global scheduling (**Paper III**) has because the DAGs are competing for the same processors. We use the global, work-conserving scheduler presented in **Paper II** to schedule the tasks of a single DAG to a cluster of unrelated processors, and we introduce a new approach to compute the makespan. We use preemptive uniprocessor EDF to schedule sequentially the tasks of multiple DAGs that are assigned to a single processor. The concept of the processor value is introduced to select the right processor type for each cluster such that all the DAGs meet their deadlines. The simulation results show that the proposed approach has a higher acceptance ratio than global scheduling on unrelated multiprocessors and than previous work that focuses on related multiprocessors.

Overall, **Paper I** shows that developing a scheduler to be time predictable by design can simplify the schedulability analysis and can provide tight makespan computations. Furthermore, from the implementation point of view, because the proposed method is simulation-based, we do not need the whole DAG stored in some data structure. We generate the DAG gradually as we simulate its execution that helps us to model large applications. The unrelated model used in **Paper II**, **Paper III**, **Paper IV** is very expressive and can model many available platforms today using a wide range of processor-types and specialized application accelerators. In addition, the unrelated model is a useful analysis tool because it is a generalization of the homogeneous and related platform models that allows us to adapt and reuse broadly used analysis tools. In **Paper II** by the modeling for unrelated multiprocessors the task-based parallel applications from the BOTS benchmark suit, we can note that the number of tasks (nodes in the DAG) is significantly larger than the number of processors that a platform may have. However, each application has very few unique tasks, i.e., tasks that perform different functions. Two tasks that are of the same unique task have the same WCET of the processors. This observation allows us to model larger applications because we could compute the parameters of the makespan by considering only the unique tasks. Also, during the actual execution of the DAG, it is probable that there are many tasks with the same processor type "preferences" (i.e., same unique task) that are executed in parallel and compete for the same processor types, which is the main idea of the greediness property used in **Paper II**, **Paper III**, **Paper IV**.

Finally, from **Paper I** we can see that the usage of a time predictable scheduler can significantly reduce the pessimism of the makespan computation. An interesting future direction would be to develop a timing anomaly-free scheduler for unrelated multiprocessors that will let us compute the makespan by simulating the execution of the DAG. After knowing how to compute a simulation-based makespan, we can develop a federated scheduling algorithm, similar to **Paper IV** to check the schedulability of multiple sporadic DAGs. A timing anomaly-free scheduler for unrelated multiprocessors is expected to be non-work-conserving so the analysis of the interference of **Paper III** cannot be used and a novel approach is needed to establish an upper bound on the interference of the DAGs that are competing for the same processors.

Bibliography

[1] Henning Butz. The airbus approach to open integrated modular avionics (ima): Technology, methods, processes and future road map. *Workshop on Aircraft System Technologies*, 2007.

[2] Leonidas Kosmidis. *Enabling caches in probabilistic timing analysis*. PhD thesis, Universitat Politècnica de Catalunya, 2018.

[3] Sujeet Milind Patole, Murat Torlak, Dan Wang, and Murtaza Ali. Automotive radars: A review of signal processing techniques. *IEEE Signal Processing Magazine*, 2017.

[4] JF Bell et al. The mars 2020 perseverance rover mast camera zoom (mastcam-z) multispectral, stereoscopic imaging investigation. *Space Science Reviews*, 2021.

[5] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011.

[6] Yale N. Patt and Sanjay J. Patel. Introduction to computing systems: From bits and gates to c and beyond. *McGraw-Hill, 2nd Edition,*, 2005.

[7] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *IEEE ISCA*, 2011.

[8] John L Hennessy and David A Patterson. *Computer architecture a quantitative approach, Sixth eition*. Elsevier, 2017.

[9] Kallia Chronaki, Miquel Moretó, Marc Casas, Alejandro Rico, Rosa M Badia, Eduard Ayguadé, and Mateo Valero. On the maturity of parallel applications for asymmetric multi-core processors. *Journal of Parallel and Distributed Computing*, 2019.

[10] John L Hennessy and David A Patterson. A new golden age for computer architecture. *Communications of the ACM*, 2019.

[11] Alba Melo, Jesus Carretero, Per Stenstrom, Sanjay Ranka, and Eduard Ayguade. Trends on heterogeneous and innovative hardware and software systems, 2019.

[12] ARM. big.little technology: The future of mobile. White paper, `https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf`, 2011.

[13] ARM. The future of compute, re-imagined. https://www.arm.com/why-arm/technologies/dynamiq, 2017.

[14] NVIDIA. Nvidia jetson agx xavier and the new era of autonomous machines. Nvidia presentation, http://info.nvidia.com/rs/156-OFN-742/images/Jetson_AGX_Xavier_New_Era_Autonomous_Machines.pdf, 2018.

[15] Eugene L Lawler and Jacques Labetoulle. On preemptive scheduling of unrelated parallel processors by linear programming. *JACM*, 1978.

[16] Björn Andersson and Gurulingesh Raravi. Real-time scheduling with resource sharing on heterogeneous multiprocessors. *Real-Time Systems*, 2014.

[17] Björn Andersson and Gurulingesh Raravi. Scheduling constrained-deadline parallel tasks on two-type heterogeneous multiprocessors. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 247–256, Brest, France, 2016. Association for Computing Machinery.

[18] Alberto Marchetti-Spaccamela, Cyriel Rutten, Suzanne Van Der Ster, and Andreas Wiese. Assigning sporadic tasks to unrelated machines. *Mathematical Programming*, 2015.

[19] Sanjoy K Baruah, Vincenzo Bonifaci, Renato Bruni, and Alberto Marchetti-Spaccamela. Ilp models for the allocation of recurrent workloads upon heterogeneous multiprocessors. *Journal of Scheduling*, 2019.

[20] Antoine Bertout, Joël Goossens, Emmanuel Grolleau, and Xavier Poczekajlo. Workload assignment for global real-time scheduling on unrelated multicore platforms. In *RTNS*, 2020.

[21] Kecheng Yang and James H Anderson. Optimal gedf-based schedulers that allow intra-task parallelism on heterogeneous multiprocessors. In *IEEE ESTIMedia*, 2014.

[22] Kecheng Yang, Ming Yang, and James H Anderson. Reducing response-time bounds for dag-based task systems on heterogeneous multicore platforms. In *ACM RTNS*, 2016.

[23] Xu Jiang, Nan Guan, Xiang Long, and Wang Yi. Semi-federated scheduling of parallel real-time tasks on multiprocessors. *IEEE RTSS*, 2017.

[24] Meiling Han, Nan Guan, Jinghao Sun, Qingqiang He, Qingxu Deng, and Weichen Liu. Response time bounds for typed dag parallel tasks on heterogeneous multicores. *IEEE TPDS*, 2019.

[25] Xuemei Peng, Meiling Han, and Qingxu Deng. Response time analysis of typed dag tasks for g-fp scheduling. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*. Springer, 2019.

[26] Meiling Han, Tianyu Zhang, Yuhan Lin, and Qingxu Deng. Federated scheduling for typed dag tasks scheduling analysis on heterogeneous multi-cores. *Journal of Systems Architecture*, 2020.

[27] Houssam-Eddine Zahaf, Zahaf Houssam-Eddine, Nicola Capodieci, Roberto Cavicchioli, Giuseppe Lipari, and Marko Bertogna. The hpc-dag task model for heterogeneous real-time systems. *IEEE Transactions on Computers*, 2020.

[28] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 1969.

[29] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *IEEE RTSS*, 1999.

[30] Jan Reineke et al. A definition and classification of timing anomalies. In *WCET*, 2006.

[31] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C Buttazzo. Response-time analysis of conditional dag tasks in multiprocessor systems. In *ECRTS*, 2015.

[32] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *IEEE RTSS*, 2012.

[33] Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor scheduling for real-time systems*. Springer, 2015.

[34] Wilhelm Reinhard et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM TECS*, 2008.

[35] Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, 2009.

[36] Theo Ungerer et al. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010.

[37] Michael Zimmer, David Broman, Chris Shaver, and Edward A Lee. Flexpret: A processor platform for mixed-criticality systems. In *RTAS*. IEEE, 2014.

[38] Christine Rochange, Pascal Sainrat, and Sascha Uhrig. *Time-Predictable Architectures*. John Wiley & Sons, 2014.

[39] Sebastian Hahn and Jan Reineke. Design and analysis of sic: A provably timing-predictable pipelined processor core. *Real-Time Systems*, pages 1–39, 2019.

[40] Alejandro Duran et al. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *ICPP*, 2002.

[41] Eduard Ayguadé et al. The design of openmp tasks. *IEEE TPDS*, 2009.

[42] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 1999.

[43] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 1988.

[44] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *IEEE ECRTS*, 2014.

[45] Sanjoy Baruah. Federated scheduling of sporadic dag task systems. In *2015 International Parallel and Distributed Processing Symposium*. IEEE, 2015.

[46] Risat Pathan, Petros Voudouris, and Per Stenström. Scheduling parallel real-time recurrent tasks on multicore platforms. *IEEE TPDS*, 2018.

[47] Niklas Ueter, Georg von der Brüggen, Jian-Jia Chen, Jing Li, and Kunal Agrawal. Reservation-based federated scheduling for parallel real-time tasks. In *IEEE RTSS*, 2018.

[48] Ashikahmed Bhuiyan, Zhishan Guo, Abusayeed Saifullah, Nan Guan, and Haoyi Xiong. Energy-efficient real-time scheduling of dag tasks. *ACM TECS*, 2018.

[49] José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Schedulability analysis of dag tasks with arbitrary deadlines under global fixed-priority scheduling. *Real-Time Systems*, 2019.

[50] Son Dinh, Christopher Gill, and Kunal Agrawal. Efficient deterministic federated scheduling for parallel real-time tasks. In *RTCSA*. IEEE, 2020.

[51] Nan Guan and Wang Yi. Fixed-priority multiprocessor scheduling: Critical instant, response time and utilization bound. In *26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 2012.

[52] Theodore P Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *RTSS*. IEEE, 2003.

[53] Marko Bertogna and Michele Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *RTSS*. IEEE, 2007.

Chapter 4

Paper I

Timing-Anomaly Free Dynamic Scheduling of Task-Based Parallel Applications

Petros Voudouris, Per Stenström, Risat Pathan

IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017.

Timing-Anomaly Free Dynamic Scheduling of Task-Based Parallel Applications

Petros Voudouris, Per Stenström, Risat Pathan
Chalmers University of Technology, Sweden
{petrosv, per.stenstrom, risat}@chalmers.se

***Abstract*—Multicore architectures can provide high predictable performance through parallel processing. Unfortunately, computing the makespan of parallel applications is overly pessimistic either due to load imbalance issues plaguing static scheduling methods or due to timing anomalies plaguing dynamic scheduling methods. This paper contributes with an anomaly-free dynamic scheduling method, called `Lazy`, which is non-preemptive and non-greedy in the sense that some ready tasks may not be dispatched for execution even if some processors are idle.**

Assuming parallel applications using contemporary task-based parallel programming models, such as OpenMP, the general idea of `Lazy` is to avoid timing anomalies by assigning fixed priorities to the tasks and then dispatch *selective* highest-priority ready tasks for execution at each scheduling point. We formally prove that `Lazy` is timing-anomaly free. Unlike all the commonly-used dynamic schedulers like breadth-first and depth-first schedulers (e.g., CilkPlus) that rely on analytical approaches to determine an upper bound on the makespan of parallel application, a safe makespan of a parallel application is computed by simulating `Lazy`. Our experimental results show that the makespan computed by simulating `Lazy` is much tighter and scales better as demonstrated by four parallel benchmarks from a task-parallel benchmark suite in comparison to the state-of-the-art.

I. Introduction

Multicore architectures can offer high and predictable performance, through parallelism, for real-time applications. The challenge, however, is to make a sufficiently tight estimate of the makespan which is the longest possible time the application may take to finish its execution. Today, parallel applications increasingly use task-based parallel programming models such as OpenMP 4.0. In task-based parallel programs, dependencies between tasks are specified by programmers. Thus, they can be viewed as a direct acyclic graph (DAG), where nodes are tasks and edges are dependencies between tasks. In deriving the makespan for such DAGs, one must take into account how tasks are scheduled onto the processors (cores) of the multicore architecture.

While scheduling of single-threaded applications on multiprocessor systems is well researched, the literature on predictable scheduling of parallel applications is sparse. First of all, while some work has targeted static scheduling, i.e., statically pre-assigning tasks to fixed cores [17], [13], [15], static scheduling fundamentally underutilizes hardware resources due to load imbalance or communication overheads. Dynamic scheduling, on the other hand, can significantly improve resource utilization. However, it suffers from *timing anomalies* [5], [9], [16] meaning that the makespan of the parallel application can be longer if the execution time of some task is shorter than its estimated worst-case execution time (WCET). Therefore, in the presence of timing anomalies, overly pessimistic assumptions have to be made to provide a safe upper bound on the makespan making it challenging to enjoy a predictable speedup on a multicore architecture. In order to determine a safe makespan for greedy dynamic scheduling algorithms, in the presence of timing anomalies, Melani et al. [11] assume that all tasks can interfere with the tasks that dictate the critical path of a parallel application making makespan estimation overly pessimistic[1].

This paper proposes, for the first time, a formally proven timing-anomaly-free dynamic scheduling algorithm — *the Lazy Scheduler* (`Lazy`) — that offers safe estimation of the makespan of parallel applications by simulating the execution of the application assuming the execution time of each task is its designated WCET. `Lazy` is priority-based, non-greedy, and non-preemptive. Each task of a parallel application is assigned a fixed priority. The highest-priority ready tasks are dispatched for execution on the idle processors based on a condition. If a ready task does not satisfy this condition, the task is not dispatched for execution even if some processor is idle (i.e., `Lazy` is non-greedy). A task that is dispatched for execution finishes its execution without any preemption (i.e., `Lazy` is non-preemptive).

In order to assess the effectiveness of `Lazy` in determining makespan of parallel applications, we study its performance in dynamic scheduling of four task-based parallel OpenMP applications from the BOTS benchmark suite [4]. These applications are widely used in many different fields of computing (e.g., data processing, sorting, scientific applications, image processing, etc.). We develop a simulation framework to determine the makespan of parallel applications using `Lazy`. Our setup is composed of two main parts: the task generator and the scheduler. The *task generator* module models the target parallel application as a DAG by dynamically generating the tasks/nodes of the target parallel applications. The *scheduler* implements `Lazy` and simulates the execution of nodes of the target application on a multicore platform to compute the makespan assuming the execution time of each task is its designated WCET.

[1]The analysis in [11] has two components: (i) analysis of a single DAG (Eq.(5)), and (ii) analysis of multiple recurrent DAGs. The former analysis is the state-of-the-art result for makespan estimation of a single DAG.

978-1-5090-5269-1/17 $31.00 © 2017 IEEE

As a baseline, we establish a safe upper bound of the makespan using the analytical approaches of [11] which is pessimistic but safely estimates the makespan of parallel application for any greedy dynamic scheduling algorithm. We use *tightness* and *scalability* as key metrics to compare effectiveness of `Lazy` in determining the makespan of parallel applications with the baseline. Tightness is defined as the ratio of makespans of the baseline and `Lazy` using the same number of processors whereas scalability compares makespans of a scheduling algorithm for two different configurations: smaller input size with smaller number of processors and larger input size with larger number of processors. The main contributions are the following

- We propose the `Lazy` scheduler which is a timing-anomaly-free dynamic scheduling method for parallel applications. We formally prove that it is timing-anomaly free. To the best of our knowledge, the `Lazy` scheduler is the first timing-anomaly-free dynamic scheduler which allows determining the makespan by simulating the execution of parallel application.
- A methodology to generate the DAG representation of OpenMP parallel application is presented. Our simulation framework can be used to empirically establish the tightness and scalability of any dynamic scheduling algorithm. It is shown that the `Lazy` scheduler is significantly tighter and more scalable than the state-of-the-art greedy methods. We find that `Lazy` scheduler is always tighter (on average 9% and maximally 36%) and always scales better (on average 14% and maximally 30%) in comparison to the baseline.

The makespan of a single DAG can be used to determine whether the DAG meets some specified deadline. This research can be applied to *federated scheduling* [8] of multiple recurrent constrained-deadline DAGs on M processing cores as follows. Each DAG is assigned a fixed number of dedicated cores on which its nodes are scheduled based on our proposed `Lazy` scheduler. The minimum number of dedicated cores to meet all the deadlines of a DAG can be determined using bisection search in the range [1, M] such that the makespan (computed using our approach) is not larger than the deadline. If the sum of required number of dedicated cores for all the DAGs is not larger than M, then we can guarantee that all the DAGs meet their deadlines.

The rest of the paper is organized as follows. First, the system model is presented in Section II. We then provide examples of how timing anomalies may occur in dynamically scheduled programs in Section III. We describe `Lazy` and prove its anomaly freedom in Section IV. The methodology to model the applications as DAG is described in Section V. The simulation framework and the evaluation metrics are presented in Section VI. Finally, related work is presented in Section VII before we conclude in Section VIII.

II. System Model

We consider a multicore architecture with M identical cores such that each core has a (normalized) speed of one. A parallel application is modeled as a directed acyclic graph (DAG) denoted $G = (V, E)$, where $V = \{v_1, \dots v_n\}$ is a set of n nodes and $E \subseteq (V \times V)$ is a set of directed edges. Each node $v_p \in V$ represents a sequential chunk of execution (called, task) having C_p equal to its WCET. If $(v_p, v_q) \in E$, then v_q can start execution after node v_p completes. As this paper focuses on makespan for parallel applications, we make the simplifying assumption that WCET of each sequential task (i.e., a node in the DAG) is known. We define a *path* of G originating at node v_a to node v_b as a sequence of nodes $(v_a, \dots v_b)$ such that $(v_j, v_{j+1}) \in E$, $a \leq j < b$.

For each node $v_j \in V$, we define the *predecessors* of v_j to be the set of nodes $v_k \in V$ such that there is an edge from v_k to v_j. For each node $v_j \in V$, we define the *successors* of v_j to be the set of nodes $v_k \in V$ such that there exists an edge from v_j to v_k. For each node $v_j \in V$, we define the *ancestors* of v_j to be the set of nodes $v_k \in V$ such that there exists a path from v_k to v_j. Similarly, for each node $v_j \in V$, we define the *descendants*, of v_j to be the set of nodes $v_k \in V$ that there is a path from v_j to v_k. These sets can be computed in linear time in the size of DAG G [3].

A node with no incoming and no outgoing edge is called a *source* and *sink* respectively. Without loss of generality we assume that there is exactly one source (denoted as v_{src}) and one sink (denoted as v_{sink}) of G. If there is more than one source/sink node, a dummy node with WCET zero as a new source/sink node is added. The terms "node" and "task" are used interchangeably.

III. Timing Anomalies

The makespan of a dynamically scheduled parallel application may increase when some tasks take less than their WCETs at run-time. This is known as an *execution-time-based timing anomaly* [5], [9], [16]. An example of such an anomaly is illustrated in Figure 1. The C_i value alongside each node is the WCET of the corresponding task in Figure 1. The DAG is executed based on a non-preemptive Breadth First Schedule (BFS) on two processors P_0 and P_1.

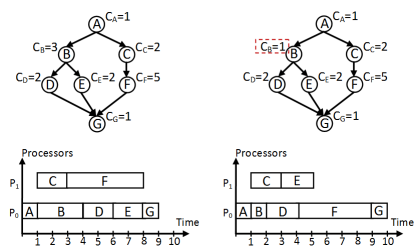


Figure 1: Execution time based anomaly.

Consider, for example, the DAG and the schedule on the left-hand side of Figure 1. The execution time of the application is 9 units. Now consider the case when node B does not execute for 3 units but finishes after 1 unit and all other nodes execute according to their WCET. The DAG and the schedule for this scenario are shown on the right-hand side of 1. The execution time of the application is 10 units. Hence, the overall execution time of the application is increased (from 9 to 10 units) when node B takes less time units than its WCET. This example demonstrates an *execution-time-based timing anomaly*.

A trivial approach to avoid such timing anomalies is as follows: each node of a DAG is forced to execute for its WCET (i.e., if it completes earlier, then it idles the core until its WCET). However, this trivial approach has several problems. *First*, the scheduler needs to know the WCETs of all the nodes during runtime and must keep track of the elapsed execution time of each node to determine if it completes earlier than its WCET. It is not clear how the WCETs (that are known offline) for all the nodes could be made available at runtime for such comparison. In addition, the tracking and comparison for a large number of nodes incur overhead. *Second*, when a node completes earlier, a timer needs to be programmed to implement the "idling of core". We may have a peculiar situation where M timers are programmed to idle all the M cores. Moreover, managing M different timers may not scale for large M. *Third*, handling the timers' interrupts also has overhead. Since the parallel applications that we consider have several thousand nodes — each of which may complete earlier — handling several thousand timers' interrupts is not practical.

IV. The Proposed Scheduler: Lazy

This section first presents a policy to assign fixed priorities to the nodes of a DAG. Second, the details of our proposed scheduler `Lazy` is presented. Finally, we prove that `Lazy` is free from any execution-time-based timing anomaly.

A. Priority Assignment Policy

Each node in the DAG is assigned a fixed priority. The fixed priority of newly generated child node is assigned based on the fixed priority of its parent. The priority of a node v_i is denoted as a pair (L_i, lp_i) where L_i is the *level* of the node in the DAG and lp_i is *level priority* of v_i at level L_i. If v_i is the root node of the DAG, then v_i is assigned level 1 and level priority 1, i.e. $(L_i, lp_i) = (1, 1)$. The successor nodes of v_i are assigned fixed priorities based on the priority of their parent.

Let the maximum number of children that are generated by any node v_i with priority (L_i, lp_i) is (called, the maximum degree) D. All the D children are considered for priority assignment based on the order in which they are created by the parent. Let the D child nodes of parent v_i are $u_1, u_2, \ldots u_D$ (ordered based on the creation order). The j^{th} child node u_j, that is generated from a parent node v_i with priority (L_i, lp_i), is assigned level $L_j = (L_i + 1)$ and level priority $lp_j = (D \times lp_i - (D-1) + j)$ for $j = 1, \ldots D$. The intuition behind computing the level priority lp_j is that the structure of the DAG might ultimately look like a completely balanced tree where each internal node has D children.

We assume that a smaller value implies higher priority. The priorities of two nodes v_i and v_k are compared as follows. First, the levels of L_i and L_k are compared. If v_i has a smaller level than v_k (i.e., $L_i < L_k$), then v_i has higher priority over v_k. If $L_i = L_k$ and $lp_i < lp_k$, then the node v_i has higher priority; otherwise, node v_k has higher priority.

Parallel tasks generated by the same parent usually need to synchronize their results (e.g., using `taskwait` pragma in OpenMP). We model such synchronization using so called *synchronization node*. Such synchronization nodes are generally the sequential bottleneck in exploiting parallelism at the higher level of a DAG. We assign special priorities to such a synchronization node v_s at the K^{th} level of the DAG. If a parent node v_i requires its child nodes to synchronize on a node v_s, then the priority (L_s, lp_s) of the synchronization node v_s is assigned as follows.

The level $L_s = K$ is the level of node v_s in the DAG and the level priority $lp_s = lp_i$ is equal to the level priority of node v_i. Since a parent node will have a lower level-priority value (i.e., higher fixed priority), the synchronization nodes at a particular level of the DAG will have higher priority over other non-synchronization nodes at the same level. This ensures that synchronization nodes are executed with higher priority to exploit parallelism further down the DAG. Priorities to the tasks can be assigned during run-time based on the policy presented above. Each node gets the same priority even if the structure of the DAG changes, for example, due to conditional nodes that may or may not be generated based on the input values. Since the priority assignment (L_i, lp_i) is unique for all the nodes, a unique single priority, denoted by p_i, can be produced where

$$p_i = (D^{L_i - 1} - 1)/(D - 1) + lp_i \tag{1}$$

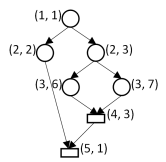


Figure 2: Example of priority assignment

For example, consider a parallel application that is implemented as a collection of parallel tasks that can be modeled as a binary tree. For such a DAG, each parent node may generate at most two children, i.e., $D = 2$. Consider a parent v_i with priority $(L_i, lp_i) = (2, 3)$. Let u_1 and u_2 are the two children of v_i. Each such child is assigned level 3 since

$(L_i + 1) = 3$. The child u_1 is assigned a level priority $(D \times lp_i - (D-1) + 1) = (2 \times 3 - (2-1) + 1) = 6$ while the other child u_2 is assigned a level priority $(D \times lp_i - (D-1) + 2) = (2 \times 3 - (2-1) + 2) = 7$. The corresponding synchronization node v_s appears at level $K = 4$, then $L_s = 4$ and $lp_i = lp_s = 3$. Therefore, u_1, u_2 and u_s are assigned priorities $(3, 6)$, $(3, 7)$ and $(4, 3)$ respectively. Figure 2 shows an example of our proposed priority assignment to a binary DAG. Note that all the nodes are assigned different priorities.

From our priority assignment policy, it can be seen that a new priority is assigned based on the priority level of the parent and the maximum degree. Consequently, our priority assignment can be used during run-time since only local information (i.e., priority information of the parent) and no global information (i.e., information about other nodes) of DAG is needed. In addition, a single-valued priority p_i can also be computed based on (L_i, lp_i) for node v_i. Such single-valued priority p_i is used by our proposed `Lazy` scheduler.

B. Scheduling Policy: `Lazy`

`Lazy` is a fixed-priority-based non-greedy and non-preemptive scheduler. In `Lazy`, the highest-priority ready tasks from the ready queue are dispatched for execution on idle processors if the condition $\mathcal{C}$ (given below) is satisfied. Before we present the details on how the `Lazy` scheduler is implemented in our simulation, we need the following definitions and notations.

A task is called active at time t if it has been released and not yet completed its execution by time t. The set of active tasks that were dispatched for execution before time t but have not yet finished their execution is denoted by set $\mathcal{EXE}_t$. Note that $\mathcal{EXE}_t$ is a subset of the set of active tasks since all the tasks that are dispatched before time t and have not completed execution by time t are also active tasks at time t. The tasks in the ready queue are those tasks that are active at time t but have not been dispatched yet (i.e., active tasks that are not in $\mathcal{EXE}_t$). We denote by $\mathcal{RQ}$ the set of tasks in the ready queue awaiting execution. Therefore, the set $(\mathcal{RQ} \cup \mathcal{EXE}_t)$ is the set of active tasks at time t.

We denote by v_t^{hpAct} the task with priority p_t^{hpAct}as the highest-priority active task in set $(\mathcal{EXE}_t \cup \mathcal{RQ})$ at time t. Note that $v_t^{hpAct} \in \mathcal{EXE}_t$ if v_t^{hpAct} was dispatched before time t; otherwise $v_t^{hpAct} \in \mathcal{RQ}$. We also denote $phc(v_t^{hpAct})$ the highest priority of any child node that is generated by node v_t^{hpAct}. For example, if node v_t^{hpAct} generates at most 2 nodes with priorities p_a and p_b where $p_a < p_b$, then $phc(v_t^{hpAct}) = p_a$ since p_a is the highest priority of any child node that is generated by node v_t^{hpAct}. The value of $phc(v_x)$ for a node v_x can be computed using the priority assignment policy in subsection IV-A.

Without loss of generality, we assume that the system starts at time $t = 0$ when all the processors are idle and the ready queue $\mathcal{RQ}$ only has the root of the DAG. The `Lazy` scheduler takes new scheduling decisions at time t when the following event **E** occurs at time t:

- **Event E:** When $t = 0$ or when some task completes its execution while the ready queue is non-empty.

The algorithm stops when all the processors become idle and the ready queue is empty. Given that an event E occurs at time t, the highest-priority active task v_t^{hpAct} at time t is determined. Recall that v_t^{hpAct} may be in set $\mathcal{EXE}_t$ or in set $\mathcal{RQ}$. Since `Lazy` is non-preemptive scheduler, task v_t^{hpAct} continues its execution if it is in set $\mathcal{EXE}_t$. Tasks from the ready queue $\mathcal{RQ}$ are dispatched by `Lazy` at time t. The `Lazy` scheduler dispatches the highest-priority task v_i with priority p_i from the ready queue $\mathcal{RQ}$ to an idle processor if the following condition $\mathcal{C}$ is satisfied:

$$\mathcal{C}\textbf{:}\ \ p_i \leq (p_t^{hpAct} + M - 1) \ \textbf{or}\ p_i < phc(v_t^{hpAct})$$

where

- p_t^{hpAct} is the priority of the highest priority active task at time t;
- M is the number of processors; and
- $phc(v_t^{hpAct})$ is the highest priority of any child node of node v_t^{hpAct}.

Condition $\mathcal{C}$ considers single-valued priority p_i that can be computed from v_i's priority (L_i, lp_i) using Eq. (1).

Algorithm 1 LazyDispatcher

```
1: procedure LazyDispatcher(List < Task > RQ)
2:     v_t^hpAct ← highest-priority task in EXE_t ∪ RQ
3:     while RQ ≠ ∅ do
4:         v_i ← highest priority task in RQ
5:         if C is true and there is an idle processor then
6:             Remove v_i from RQ
7:             Dispatch v_i to an idle processor
8:         else
9:             Exit
```

The detailed functionality of the `Lazy` scheduling policy is presented in Algorithms 1. `Lazy` takes as input the $\mathcal{RQ}$ and it dispatches new tasks to idle processors based on condition $\mathcal{C}$. Algorithm 1 at line 2 determines the highest-priority active task v_t^{hpAct} which may be in $\mathcal{EXE}_t$ or $\mathcal{RQ}$. The task v_t^{hpAct} can be determined by comparing the priorities of the highest-priority tasks in each set $\mathcal{EXE}_t$ and $\mathcal{RQ}$. The while loop in line 3 iterates as long as there is at least one ready task in the $\mathcal{RQ}$. During each iteration of this while loop, the highest-priority task from the ready queue $\mathcal{RQ}$ is stored in variable v_i in line 4. The highest-priority ready task v_i can be dispatched for execution on an idle processor if the condition $\mathcal{C}$ in line 5 is true.

If the condition in line 5 is true (i.e., task v_i satisfies $\mathcal{C}$ and there is an idle processor), then task v_i is removed from the ready queue $\mathcal{RQ}$ (line 6) and dispatched for execution on an idle processor (line 7). If the condition in line 5 is

false, then the while loop is exited (line 9). In such case no more tasks from the ready queue is dispatched for execution even if some processor is idle, which is essential to prove that Lazy is a timing-anomaly free scheduler (Theorem 1). After the while loop from line 9 is exited, Lazy waits for a task-completion event where the ready queue $\mathcal{RQ}$ is updated with new nodes (if generated).

The Lazy scheduler is non-preemptive since a task from the ready queue is dispatched only if there is an idle processor. The salient feature of condition $\mathcal{C}$ is that a lower-priority task v_i is possible to be dispatched if there are enough processors to execute all the higher priority tasks that are active at time t or may become active after time t. This crucial feature of Lazy ensures that a lower-priority task v_i during actual runtime cannot start its execution later than the time that is considered when estimating (offline) the makespan of the DAG because enough cores are reserved for all of its higher-priority tasks. Based on this feature, we will formally prove that Lazy is timing-anomaly free.

C. Proof of execution-time anomaly freedom

To prove that Lazy is a timing anomaly free scheduler in Theorem 1, we need Lemma 1–Lemma 3.

Lemma 1. *If node v_a is an ancestor of node v_c, then the priority of node v_a is higher than the priority of v_c, i.e., $p_a < p_c$ for our priority assignment policy.*

Proof: Let $v_1 = v_a, v_2, \dots v_k = v_c$ be a chain of nodes such that v_{i-1} is a parent node of v_i for $i = 2, 3, \dots k$. The j^{th} child of node v_i has priority $lp_j = (D \times lp_i - (D-1) + j)$. Since $j \geq 1$, $D \geq 1$, we have that $D \times lp_i - (D-1) + j > lp_i$. Therefore, the child has lower priority than the priority of its parent. Since v_{i-1} is a parent of v_i in the path, we have $p_{i-1} < p_i$ for $i = 2, 3, \dots k$ based on Eq. (1). Consequently, $p_1 = p_a < p_2 < p_3 < \dots p_k = p_c$. Therefore, any ancestor of v_c has higher priority than v_i. ■

For the remainder of this section consider the time instants in set $\{t_1, t_2, \dots t_e\}$ at each of which at least one task completes during the execution of a DAG in Lazy scheduling such that $t_1 = 0$, $t_1 < t_2 \dots t_e$, and no task completes execution in (t_i, t_{i+1}) for $i = 1, 2, \dots (e-1)$. Also assume that node $v_i = v_t^{hpAct}$ is the highest-priority active node at time t_i for $i = 1, 2, \dots e$.

Lemma 2. *The priority of each node is unique for our priority assignment policy.*

Proof: New child nodes are generated only if some task completes execution (i.e., event E occurs). We will prove this lemma using induction on t_k for $k = 1, 2, \dots e$.

The node that is generated at time t_1 is the root node with priority 1. Since there is only one root node, all the priorities of the nodes that are generated by time t_1 are unique. Assume that the priorities of all the tasks generated up to time t_{j-1} are unique for some j where $j \leq e$. We will show that all the priorities of the tasks generated up to time t_j are also unique. Note that no new node is generated in the interval (t_a, t_{a+1}) because no task completes execution in (t_a, t_{a+1}). After time instant t_{j-1}, new nodes can only be generated at time t_j from the nodes that complete execution at time t_j. Let v_i be such a node that completes execution at time t_j.

According to our priority assignment policy, the x^{th} and y^{th} children of node v_i have level priorities $lp_x = (D \times lp_i - (D-1) + x)$ and $lp_y = (D \times lp_i - (D-1) + y)$, respectively. Since $D \geq x \geq 1$, $D \geq y \geq 1$ and $x \neq y$, the followings are true:

$$\begin{aligned} &lp_i \neq D \times lp_i - (D-1) + x \\ &lp_i \neq D \times lp_i - (D-1) + y \\ &D \times lp_i - (D-1) + x \neq D \times lp_i - (D-1) + y \end{aligned} \tag{2}$$

Therefore, the parent v_i and the children nodes of v_i have unique priorities. Since the nodes that generate children at time t_{j-1} have unique priorities (inductive hypothesis), the children generated by two different parents v_q and v_r with priority p_q and p_r at time t_j also have different priorities (follows from Eq. (2)). Therefore, all the nodes generated by time t_j have unique priorities. This lemma also holds for singled-valued priorities that are computed using Eq. (1). ■

Lemma 3. *All the tasks having priorities (strictly) higher than p_i must have completed their execution by time t_i in Lazy scheduling where v_i is the highest priority active task at time t_i for $i = 1, 2 \dots e$.*

Proof: We use induction on t_i to prove this lemma. This lemma trivially holds for time instant t_1 when the system starts. The highest-priority active task at time t_1 is the root node of the DAG and there exists no task having priority higher than that of the root node (according to Lemma 1).

Assume that the lemma holds for time t_{i-1}. We will show that the lemma also holds for time t_i. Since the lemma holds for time t_{i-1}, all the nodes with priority higher than p_{i-1} must have completed by time t_{i-1} (inductive hypothesis). Recall that the highest-priority active node at time t_{i-1} and t_i are v_{i-1} and v_i, respectively. Assume a contradiction that all the nodes with priority higher than p_i have not completed by time t_i. In particular, there is node v_h with priority higher than v_i that has not completed its execution by time t_i. Since v_h has higher priority than that of v_i, we have $p_h < p_i$.

A node remains active until it completes its execution. Since node v_h has not completed its execution by time t_i, we must have either (i) v_h is active at time t_i, or (ii) v_h becomes active after time t_i. If v_h is active at time t_i, then v_i cannot be the highest-priority active node at time t_i since $p_h < p_i$ (a contradiction). If v_h becomes active after time t_i, then there is at least one of the ancestors of v_h that has not completed its execution by time t_i. If all the ancestors of v_h have completed their execution by time t_i, then v_h

cannot become active after time t_i. This is because a node becomes active when all its ancestors complete execution. In other words, at least one of the ancestors of v_h is active at time t_i. Let v_a be the ancestor of v_h that is active at time t_i. Since an ancestor has higher priority than each of its descendants according to Lemma 1, we have $p_a < p_h$. Since $p_h < p_i$, we have $p_a < p_i$ while both v_a and v_i are active at time t_i. Therefore, v_i cannot be the highest-priority active node at time t_i (a contradiction). Therefore, all active nodes with priorities higher than v_i must have completed their execution by time t_i. ■

Theorem 1. *The* `Lazy` *scheduler is a timing anomaly free scheduler.*

Proof: Let $\mathcal{W}_S$ be the schedule that the `Lazy` scheduler generates when each task of a DAG executes for its WCET. Let $\mathcal{O}_S$ be such an (arbitrary) schedule that the `Lazy` scheduler generates when some task of the DAG executes **less** than its WCET. We assume that the execution of the root node starts at time zero in both $\mathcal{W}_S$ and $\mathcal{O}_S$. We will show that no task in $\mathcal{O}_S$ finishes later than the time when that task finishes is $\mathcal{W}_S$, which implies that `Lazy` is a timing-anomaly free dynamic scheduler.

Let $s_i^{\mathcal{W}}$ and $s_i^{\mathcal{O}}$ denote the time instants when node v_i starts execution in $\mathcal{W}_S$ and $\mathcal{O}_S$, respectively. Similarly, let $f_i^{\mathcal{W}}$ and $f_i^{\mathcal{O}}$ denote the time instants when node v_i finishes its execution in $\mathcal{W}_S$ and $\mathcal{O}_S$, respectively. Without loss of generality assume that all the nodes $v_1, v_2 \dots v_n$ of the DAG are indexed in decreasing priority order such that $p_1 < p_2 < p_3 \dots < p_n$. Let C_i and C_i', where $C_i' \leq C_i$, be the execution time of node v_i respectively in $\mathcal{W}_S$ and $\mathcal{O}_S$ for each $i = 1, 2, \dots n$. In other words, each node takes its WCET in $\mathcal{W}_S$ while it may take less than its WCET in $\mathcal{O}_S$. Since `Lazy` is non-preemptive, the following holds for all $i = 1, 2, \dots n$:

$$f_i^{\mathcal{W}} = s_i^{\mathcal{W}} + C_i \tag{3}$$

$$f_i^{\mathcal{O}} = s_i^{\mathcal{O}} + C_i' \leq s_i^{\mathcal{O}} + C_i \tag{4}$$

We will use mathematical induction on the index (i.e., priority) of the nodes to show that $s_i^{\mathcal{O}} \leq s_i^{\mathcal{W}}$ for $i = 1, 2, \dots n$. Then it follows from Eq. (3) and Eq. (4) that $f_i^{\mathcal{O}} \leq f_i^{\mathcal{W}}$ for $i = 1, 2, \dots n$.

Base Case ($v_i = v_1$): Since the root node v_1 always starts execution at time zero in both $\mathcal{W}_S$ and $\mathcal{O}_S$, we have $s_i^{\mathcal{O}} = s_i^{\mathcal{W}} = 0$. Therefore, $s_i^{\mathcal{O}} \leq s_i^{\mathcal{W}}$ for $i = 1$.

Inductive Step: Assume that $s_j^{\mathcal{O}} \leq s_j^{\mathcal{W}}$ for all $j = 1, 2, \dots (i-1)$. We will show that $s_i^{\mathcal{O}} \leq s_i^{\mathcal{W}}$.

Since $s_j^{\mathcal{O}} \leq s_j^{\mathcal{W}}$ for all $j = 1, 2, \dots (i-1)$, it follows from Eq. (3) and Eq. (4) that

$$f_j^{\mathcal{O}} \leq f_j^{\mathcal{W}} \tag{5}$$

Let $act_i^{\mathcal{W}}$ and $act_i^{\mathcal{O}}$ denote the time instants when node v_i becomes active respectively in $\mathcal{W}_S$ and $\mathcal{O}_S$. We will first show that $act_i^{\mathcal{O}} \leq act_i^{\mathcal{W}}$. A node becomes active when all its predecessors complete execution in `Lazy` scheduling. Note that the predecessors of node v_i have higher priorities than the priority of v_i according to Lemma 1. Since $f_j^{\mathcal{O}} \leq f_j^{\mathcal{W}}$ from Eq. (5), all the higher priority tasks (including the predecessors) of v_i complete their execution in $\mathcal{O}_S$ no later compared to that of in $\mathcal{W}_S$. Therefore, node v_i in $\mathcal{O}_S$ becomes active no later than the time when it becomes active in $\mathcal{W}_S$. Therefore, $act_i^{\mathcal{O}} \leq act_i^{\mathcal{W}}$. Since a node cannot start its execution until it becomes active, we have $act_i^{\mathcal{W}} \leq s_i^{\mathcal{W}}$ and $act_i^{\mathcal{O}} \leq s_i^{\mathcal{O}}$. Because $act_i^{\mathcal{O}} \leq act_i^{\mathcal{W}}$. Therefore, we have

$$act_i^{\mathcal{O}} \leq act_i^{\mathcal{W}} \leq s_i^{\mathcal{W}} \tag{6}$$

Consider the time instant T in the $\mathcal{O}_S$ schedule such that $T = s_i^{\mathcal{W}}$. An active node remains active until it completes its execution. We will show that $s_i^{\mathcal{O}} \leq s_i^{\mathcal{W}}$ based on two cases: case (i) node v_i is not active at time $T = s_i^{\mathcal{W}}$ in $\mathcal{O}_S$; case (ii) node v_i is active at time $T = s_i^{\mathcal{W}}$ in $\mathcal{O}_S$.

Case (i) If node v_i is not active at time $T = s_i^{\mathcal{W}}$ in $\mathcal{O}_S$, then node v_i has already completed its execution in $\mathcal{O}_S$ since it has become active in $\mathcal{O}_S$ no later than time $T = s_i^{\mathcal{W}}$ according to Eq. (6). Therefore, $f_i^{\mathcal{O}} \leq s_i^{\mathcal{W}}$. From Eq. (4), we also have $s_i^{\mathcal{O}} \leq f_i^{\mathcal{O}}$. Consequently, $s_i^{\mathcal{O}} \leq s_i^{\mathcal{W}}$.

Case (ii): In such case, node v_i is active at time $T = s_i^{\mathcal{W}}$ in $\mathcal{O}_S$. We will show that node v_i in such case starts no later than time $T = s_i^{\mathcal{W}}$ in $\mathcal{O}_S$.

Note that node v_i is also an active node at time $s_i^{\mathcal{W}}$ in $\mathcal{W}_S$ since it starts execution at time $s_i^{\mathcal{W}}$ in $\mathcal{W}_S$. In addition, an event E occurs at time $s_i^{\mathcal{W}}$ in $\mathcal{W}_S$ since a scheduling decision (i.e., dispatching of v_i) is taken at time $s_i^{\mathcal{W}}$. Let v_h be the highest-priority active task in $\mathcal{W}_S$ at time $s_i^{\mathcal{W}}$ when node v_i is dispatched for execution in $\mathcal{W}_S$.

According to Lemma 3, all the tasks having priorities higher than that of v_h must have completed their execution in $\mathcal{W}_S$ by time $s_i^{\mathcal{W}}$. Since $f_j^{\mathcal{O}} \leq f_j^{\mathcal{W}}$ for $j = 1, 2, \dots (i-1)$ according to Eq. (5) and v_h's priority is not smaller than the priority of v_i, all the tasks having priorities higher than that of v_h must have completed their execution also in $\mathcal{O}_S$ by time $T = s_i^{\mathcal{W}}$. In other words, the set of active tasks at time $T = s_i^{\mathcal{W}}$ in $\mathcal{O}_S$ must have priorities lower or equal to v_h.

Therefore, the following two properties A1 and A2 hold for time $T = s_i^{\mathcal{W}}$ in both $\mathcal{O}_S$ and $\mathcal{W}_S$ schedules:

> (A1) Node v_i is active, and (A2) The set of active tasks have priorities lower than or equal to v_h.

Since v_i was dispatched at time $s_i^{\mathcal{W}}$ in $\mathcal{W}_S$ where both A1 and A2 hold at time $s_i^{\mathcal{W}}$, condition $\mathcal{C}$ was satisfied for v_i at time $s_i^{\mathcal{W}}$ in $\mathcal{W}_S$ as a prerequisite for task v_i to be dispatched according to line 5 in the algorithm in Figure 1. If an event E also occurs at time $T = s_i^{\mathcal{W}}$ in $\mathcal{O}_S$, then node v_i will not be scheduled later than time $T = s_i^{\mathcal{W}}$ since condition $\mathcal{C}$ must be satisfied at the latest at time $T = s_i^{\mathcal{W}}$ in $\mathcal{O}_S$.

If an event E does not occur at time $T = s_i^{\mathcal{W}}$ in $\mathcal{O}_S$, then property A1 and A2 must hold at an earlier time instant T_x,

where $T_x < T = s_i^{\mathcal{W}}$, in $\mathcal{O}_S$ such an event E occurs at time T_x in $\mathcal{O}_S$. After all the higher-priority active ready tasks of v_i at time T_x are scheduled, there is at least one processor idle on which node v_i will be scheduled (because A1 and A2 hold at time T in $\mathcal{W}_S$ and hold at time T_x in $\mathcal{O}_S$). In such case, node v_i must have been scheduled no later than time T_x since A1 and A2 hold at time T_x (i.e., condition $\mathcal{C}$ will be satisfied the latest at time T_x in $\mathcal{O}_S$). Therefore, node v_i is dispatched for execution at the latest by time $T = s_i^{\mathcal{W}}$ if an event E occurs at time T or at the latest by time $T_x < T$ where an event E occurs at time T_x such that A1 and A2 hold at time T_x. Consequently, $s_i^{\mathcal{O}} \leq s_i^{\mathcal{W}}$ for this case. From Eq. (3) and Eq. (4) follows that

$$f_i^{\mathcal{O}} \leq f_i^{\mathcal{W}} \tag{7}$$

Therefore, it is proved that no node in $\mathcal{O}_S$ completes later than the time when it completes in $\mathcal{W}_S$. Since this is also true for the sink node, the sink node in $\mathcal{O}_S$ never completes later than the time when it completes in $\mathcal{W}_S$. Since $\mathcal{W}_S$ is the schedule generated by `Lazy` assuming each node takes its WCET to finish its execution, the completion time of an application cannot become larger if some node takes less than its WCET. Therefore, `Lazy` is timing-anomaly free. ■

V. DAG Modeling Methodology

Instead of synthetically generating some arbitrary DAGs for our experimental evaluation, we model the worst case DAG (WCDAG) of four parallel OpenMP applications from the BOTS benchmark suite [4]: Fibonacci, Sort, Strassen, and FFT. The application *Fibonacci* is a recursive parallel application and is a good representative of many recursive applications where the parallel tasks form a tree-like structure. The application Sort is used in almost all fields of computing, for example, data processing applications. The application Strassen is a very popular and efficient technique for matrix multiplication that is commonly used in many scientific applications. Finally, FFT is widely used in signal processing and image processing domains.

We assume that the applications are executed using OpenMP flag "untied" [1], so a task can be executed on any processor. The analysis of the OpenMP code for each of the applications is performed manually to model the DAG, and then implemented in our simulation framework to automatically generate the WCDAG for any input. Note that the nodes of the WCDAG are generated online when some nodes finish their execution. This is because the child nodes are generated in OpenMP dynamically during runtime based on "pragmas" that are inserted in the code.

The first step of our manual analysis to model the WCDAG is to identify the (i) segments of the code that generate the parallel tasks, (ii) the segments of the code that perform the actual work, and finally, (iii) the segments of the code that force synchronization among the generated tasks. The outcome of this analysis is the three following types of nodes that we use to model the WCDAG of an application:

- **Spawn node**: It models the cost of generation of the parallel work. For example, the *#omp pragma task* in a loop essentially generates multiple parallel tasks. If the loop iterates 5 times, then the spawn node models the cost of the generating 5 parallel tasks.
- **Basic Node**: It models the execution time of a sequentially-executed piece of code. In other words, it models the actual work of the parallel application.
- **Synchronization Node**: It models the cost of synchronization of the parallel tasks. All the (previously generated) tasks corresponding to a synchronization node need to complete their execution before the synchronization node is allowed to execute. We use synchronization nodes to model the *#omp pragma taskwait*.

The purpose of categorizing the three types of nodes mentioned above is to distinguish the different parts of the parallel application in order to model its functionality (e.g., dependencies) and the cost of generating parallel tasks and synchronization due to *taskwait* pragmas.

The DAG models of Fibonacci, Strassen and FFT do not depend on the actual *input data* but only on the *input size*. Therefore, the DAG that is generated for any of these three applications is always the same for the same input size. The WCDAG for such application is the DAG that is generated only based on specific input size. In contrast, applications Sort is composed of two parallel functions `Quicksort` and `Mergesort`. The DAG model of `Quicksort` depends only on the input size (i.e., number of elements to sort) whereas the DAG model of `Mergesort` depends not only on the input size but also on the values of the inputs (i.e., values of the elements to sort). Therefore, the DAG that is generated for the same input size but with different values of the elements to sort may be different. Such data-dependent DAG is known as conditional DAG [2], [11]. The model of the WCDAG of Sort for the same input size needs to consider the DAG among all the possible DAGs that may be generated for the same input size but with different values of the input elements. Baruah et al. [2] proposed an approach to transform a conditional DAG to a non-conditional (i.e., worst-case) DAG. We apply similar approach from [2] to model the WCDAG of application Sort.

An example of how the DAG is generated for the Algorithm 2 is presented below in Figure 3. Initially, we manually inspect the OpenMP code and model the DAG of an application as a C++ function that is used in our experiment to generate the DAG for any input. Algorithm 2 presents the pseudocode of an OpenMP recursive procedure, called f, with one branch (i.e., if-else condition in line 3), one for loop (line 6) and takes the size (i.e., length of an array) of its input parameter along with other parameters.

A corresponding C++ function, called Dag_Gen(f), is implemented to generate the DAG of the OpenMP procedure f

Algorithm 2 Pseudocode of an OpenMP procedure

```
1: procedure f(InputSize, otherParameters)
2:     threshold = 3
3:     if InputSize ≤ threshold then
4:         Some sequential computation
5:     else
6:         for i = 1; i ≤ 3; i++ do
7:             #omp pragma task
8:             f(InputSize − i, ...)
9:         #omp pragma taskwait
```

for any value of $InputSize$. The main idea of implementing the Dag_Gen(f) function is depicted in Figure 3 that shows the DAG construction of f for $InputSize = 5$.

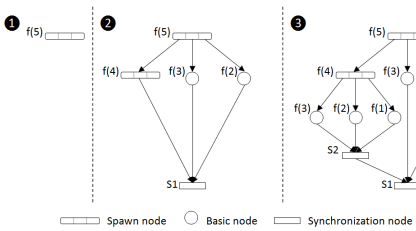


Figure 3: DAG generation for Algorithm 2

① Since the $InputSize = 5 > threshold = 3$ (i.e., the base condition in line 3 is false), the loop in line 6 is executed. The Dag_Gen(f) function models the header of the loop as a spawn node "f(5)" in Figure 3. ② The Dag_Gen(f) function then generates three new nodes "f(4)", "f(3)" and "f(2)" as the children of the spawn node "f(5)" with a corresponding synchronization node "S1". The first child node "f(4)" is a new spawn node since the base condition will not be fulfilled when we invoke f with $InputSize = 4$. The second and third child nodes, denoted as "f(3)" and "f(2)", are basic nodes since the base condition in line 3 will be satisfied. Appropriate edges are added by connecting the spawn node "f(5)" with all three children "f(4)", "f(3)", and "(f2)" which in turn connect the synchronization node "S1" to reflect the dependencies. ③ The Dag_Gen(f) function recursively generates three basic nodes "f(3)", "f(2)" and "f(1)" from the spawn node "f(4)" with a corresponding synchronization node "S2". Edges are updated and new edges are added to reflect the new dependency structure of the DAG.

VI. Simulation framework and metrics

This section presents the simulation framework and the results of the effectiveness of the `Lazy` scheduler considering the WCDAG model of applications Fibonacci, Sort, Strassen, and FFT. The simulator used in our experiments is introduced in VI-A. Two metrics (i.e., tightness and scalability) and the configurations to measure them are presented in Sections VI-B–VI-E, respectively.

A. Simulator

The simulator is event based, where an event is considered the completion of a task. Figure 4 presents an abstract view of the simulator. It takes as input, the number of processors M, the root node of the DAG model of an application, the node-generation module for different applications (denoted as "App"), the size of the input (denoted as "Input Size") of the application under study, and the scheduling policy. It returns the makespan of the application under some given scheduling policy. The "Generate Nodes" component generates the new nodes and inserts them in the $ReadyQ$. Next, based on the scheduling policy the "Scheduler" component selects the appropriate nodes and inserts them in the $RunningQ$. The maximum size of the $RunningQ$ is fixed and equal to the number of processors. The node with the minimum remaining execution time of the running nodes is considered to be completed and is removed from the $RunningQ$ and inserted in the $FinishedQ$ by the "Select Finished" component. If multiple nodes are finished at the same time, they are all inserted in the $FinishedQ$. The finished nodes are fed back to the "Generate Nodes" component in order to release the dependencies due to the completion of the tasks and progress to the generation of new nodes. When all the tasks of an application complete execution, the finish time of the last task is the makespan reported by our simulator (denoted by T_M^{Lazy}).

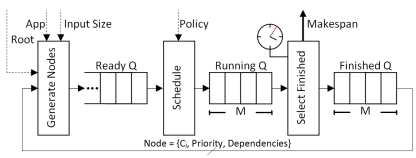


Figure 4: High level view of the experimental framework

`Lazy` is a fixed-priority-based scheduler that dispatches the highest-priority ready node on an idle core if condition $\mathcal{C}$ is satisfied. Therefore, the complexity of the runtime dispatcher of `Lazy` is similar to that of a global fixed-priority scheduler [11]. Because `Lazy` is a non-preemptive scheduler, the total number of scheduling decisions is upper bounded by the number of nodes in a DAG. In our experiments, the simulation of `Lazy` finds the makespan in few seconds for each configuration considered in this paper.

B. Configuration to Measure Tightness

To the best of our knowledge there exists no timing-anomaly free dynamic scheduler for parallel application that we can use to determine the makespan by simulating the execution of the DAG and compare it with the makespan

determined using the proposed Lazy scheduler. An analytical formula, given in Eq. (8), is derived in a recent work [11] to determine an upper bound on the makespan of a parallel application for any greedy dynamic scheduler:

$$T_M^{Greedy} = T_\infty + (T_1 - T_\infty)/M \quad (8)$$

where T_M^{Greedy} is the (upper bound on) makespan of a target application scheduled using any greedy (i.e., work-conserving) dynamic scheduler on M processors; T_∞ is the makespan of the application on infinite number of processors (i.e., it is the length of the longest path), and T_1 is the makespan of the application on one processor (i.e., it is the total work of the application). The parameter T_1 represents the sum of the WCETs of all the tasks in the application while the parameter T_∞ represents the maximum sum of the WCETs of the tasks in any source-to-sink path, called the length of the longest/critical path in the DAG. The values of T_1 and T_∞ can be computed in linear time in the representation of the DAG [3].

The term $(T_1 - T_\infty)/M$ in Eq. (8) is the maximum delay (i.e, interference) that the nodes in the critical path suffer due to the execution of the nodes that are not in the critical path. Therefore, the nodes of the critical path finish their execution no later than $T_\infty + (T_1 - T_\infty)/M$, which is the makespan of the application given in Eq. (8).

To compare the makespan of a parallel application using Lazy with that of in Eq. (8), we use a metric called tightness. We define *tightness* as the ratio between the makespan derived using Eq. (8) and the makespan generate by simulating our proposed Lazy scheduler, denoted as T_M^{Lazy}, where both consider the same number of processors i.e., tightness of Lazy is T_M^{Greedy}/T_M^{Lazy}.

It is well-know that determining the optimal (i.e., safe and the shortest) makespan of parallel applications with precedence constraints is intractable. However, the length of the optimal makespan cannot be smaller than T_∞ since nodes in the longest path must execute sequentially regardless of the number of processors. Moreover, the length of the optimal makespan cannot be smaller than T_1/M since T_1 amount of work cannot be completed less than T_1/M time units on M processors. We denote the *lower bound* on the length of the optimal schedule by T_M^{LBOPT} where $T_M^{LBOPT} = max\{T_\infty, T_1/M\}$, which is smaller than or equal to the optimal makespan. The ratio T_M^{Lazy}/T_M^{LBOPT} provides an estimate of tightness of the (hypothetical) optimal scheduler in comparison to Lazy scheduler.

Table I: Configurations for tightness

	Fib	Sort	Strassen	FFT
Input	20	32768	512	8192
$\#Nodes$	32836	16043	22410	23748
T_1	8756400	4403300	7843300	6221400
T_∞	8000	14900	2500	51020
Maximum level	39	71	12	137

To determine tightness (i.e., T_M^{Greedy}/T_M^{Lazy} and T_M^{Lazy}/T_M^{LBOPT}), we use the configurations presented in Table I for applications Fibonacci, Sort, Strassen and FFT. The second row is the input size. For Fibonacci, the input size is also the actual input but for the other applications it is the size (i.e., length) of the input array. Furthermore, the number of nodes that are generated for the different applications are presented in the third row. The WCET of the spawn, base and synchronization nodes are set to 300, 400 and 100 time units respectively. The values of T_1 and T_∞ are shown for all the applications in the third and fourth rows in Table I. The fifth row shows the maximum level of the DAG of each application.

We believe the structures of the DAGs of the four applications that we consider in Table I are diverse enough to compare our approach with the state-of-the-art in [11]. For example, the DAG of application "Strassen" has a balanced 7-ary-tree like structure while the DAG of "Sort" has a combination of balanced 4-ary-tree (quicksort part) and unbalanced 2-ary-tree (mergesort part) like structures. The DAGs of these applications in general have much larger depth and much higher total number of nodes in comparison to the synthetic DAGs evaluated in [11]. An accurate description of these benchmarks is in [4].

C. Results: Tightness

Figure 5 presents the results for tightness for all the applications. The horizontal axis is the number of processors for each application. The vertical axis is the tightness. The left bar for each application considering a particular number of processors represents the tightness of the makespan determined by Lazy with respect to Eq. (8) for any greedy scheduler (i.e, value of T_M^{Greedy}/T_M^{Lazy}). The right bar represents the tightness of the lower bound on the optimal (the shortest) makespan with respect to Lazy scheduler (i.e., value of T_M^{Lazy}/T_M^{LBOPT}).

A first observation is that, for all the cases, the estimation of makespan of the Lazy scheduler is tighter (i.e., $T_M^{Greedy}/T_M^{Lazy} \geq 1$) compared to Eq. (8) for each application. The worst-case assumption in deriving Eq. (8) is that the nodes that are not in the critical path do not run in parallel (i.e., always interfere) with the nodes of the critical path. However, the structure of a DAG may allow the nodes in the critical path to execute in parallel with nodes that are not part of the critical path. Consequently, the estimation of the makespan of a parallel application using Eq. (8) is more pessimistic than that of Lazy.

Second, it can be seen that for up to 16 processors, the tightness T_M^{Greedy}/T_M^{Lazy} of Lazy is close to 1. This is because both the Lazy scheduler and the greedy scheduler (Eq. (8)) has performance very close to the optimal scheduler. This can be verified by observing that the tightness of T_M^{Lazy}/T_M^{LBOPT} is also very close to 1. When the number of processors is small, all the processors are almost always

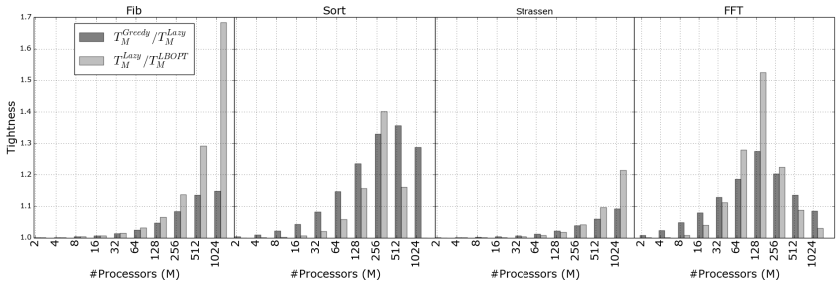


Figure 5: Tightness for Fibonacci, Sort, Strassen and FFT. Tightness with respect to T_M^{Greedy} (left bar) shows the improvement compare to state-of-the-art and the tightness with respect to T_M^{LBOPT} (right bar) shows the potential for further improvement.

busy executing some nodes of the DAG since $T_1/M >> T_\infty$ for all the applications as can be seen in Table I. Since the amount of work normalized by the number of processors is significantly larger than the length of the longest path, the processors always have some node to execute when the number of processors is relatively small. Consequently, there is very little room for further improvement for the case when the number of processors is small and the `Lazy` scheduler performs as good as the (hypothetical) optimal scheduler.

Finally, it can be observed that the tightness of `Lazy` (the left bar) increases with the increase in the number of processors for all the applications, up to the point where the length of the longest path (T_∞) is equal to the estimated makespan. Adding more processors will not finish the execution earlier than the length of the longest path. Note that the tightness of the lower bound of the optimal makespan increases with the number of processors up to the point where the makespan becomes equal with the T_∞, which means that there may be room for further improvement if T_M^{LBOPT} is very close to the actual makespan. From our experiments, we have noted that the priority assignment influences significantly the performance of the scheduler. A priority assignment which can relax some the restrictions that `Lazy` imposes while avoiding timing anomalies can improve the performance for large number of processors.

In summary, for different applications and different number of processors, the simulation of `Lazy` scheduler achieves on average 9% and a maximum of 36% tighter estimation of the makespan in comparison to the state-of-the-art in Eq. (8).

D. Configuration to Measure Scalability

To evaluate the scalability of the scheduler, the Gustafson's Law [6] is used, where the goal is to evaluate if the execution time will be invariant as the problem size and the number of processors increases. With this Law one can evaluate whether the larger problem on a larger number of processors can be executed in the same amount of time. Given some base configuration, we need to find other configuration by varying the input size of the application and the number of processors so that the ratios of workload and the number of processors for both configurations are equal. However, based on the characteristics of the applications it is very challenging to generate workload of an application to maintain such proportion exactly.

We approximate the proportional increase of the workload with respect to the workload of a base configuration, denoted by $(T_{1,base}, M^{base})$, where $T_{1,base}$ is the estimated makespan of the application on $M^{base} = 1$ processor. To generate a target workload T for a new configuration i, we increase the input size of the application to generate workload $T_{1,i} \approx T$ by executing it on one processor and we set $\lceil T_{1,i}/T_{1,base} \rceil = M^i$ as the number of processors for the i^{th} configuration $(T_{1,i}, M^i)$. The scaled speedup for `Lazy`, any greedy scheduler, and infinite number of processors is given by equations (9), (10) and (11) respectively.

$$S_i^{Lazy} = T_{M,i}^{\texttt{Lazy}}/T_{1,base} \quad (9)$$

$$S_i^{Greedy} = T_{M,i}^{Greedy}/T_{1,base} \quad (10)$$

$$S_i^{\infty} = T_{\infty,i}/T_{1,base} \quad (11)$$

where $T_{M,i}^{\texttt{Lazy}}$, $T_{M,i}^{Greedy}$ and $T_{\infty,i}$ are the makespans of an application using `Lazy` scheduler, any greedy scheduler, and the length of the longest path for the i^{th} configuration.

The outcome of the configurations for the Fibonacci, Sort, Strassen and FFT are presented in Table II. Note that, for Strassen, we are using up to seven configurations because the workload of Strassen increases at a much faster rate compared to other applications (for the seventh configuration more than 1 million nodes are generated for Strassen).

E. Results: Scalability

Figure 6 presents the results for scalability of the applications. The horizontal axis presents the different configuration points from Table II and the vertical axis shows the normalized scalability (i.e., values of S_i^{Lazy}, $S_i^{Greedy}$and S_i^{∞}). The different lines present the S_i^{Lazy} , S_i^{Greedy} and S_i^{∞} .

It is evident that for all the applications, the `Lazy` scales

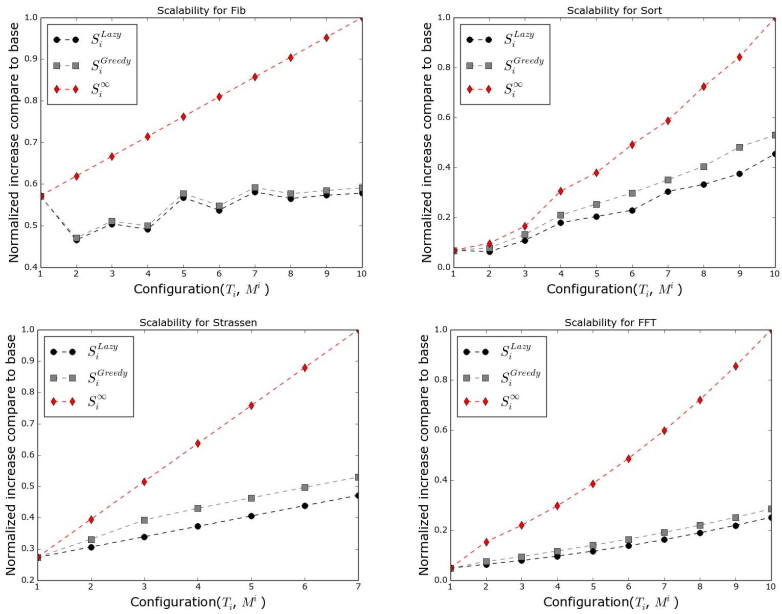


Figure 6: Scalability for Fibonacci, Sort, Strassen and FFT

Table II: Configuration ($T_{1,i}$, M^i) for scalability where $T_{1,i}$ is the workload in time units of an application on one processor and M^i is the number of processors for the i^{th} configuration to measure scalability

	Fib			Sort		
	$T_{1,i}$	P^i	$\#Nodes$	$T_{1,i}$	P^i	$\#Nodes$
1	186000	1	697	2700	1	9
2	301200	2	1129	4500	2	15
3	487600	3	1828	16900	6	59
4	789200	5	2959	92900	30	331
5	1277200	7	4789	172900	49	619
6	2066800	12	7750	473700	147	1707
7	3344400	18	12541	896100	249	3243
8	5411600	30	20293	2304100	694	8363
9	8756400	48	32836	4403300	1212	16043
10	14168400	77	53131	10854500	3207	39595
	Strassen			FFT		
	$T_{1,i}$	P^i	$\#Nodes$	$T_{1,i}$	P^i	$\#Nodes$
1	3300	1	10	14820	1	78
2	22900	8	66	71640	5	321
3	160100	52	458	151400	10	656
4	1120500	361	3202	319640	20	1344
5	7843300	2527	22410	673560	41	2756
6	54902900	17686	156866	1416280	86	5652
7	384320100	123800	1098058	2971480	178	11588
8				6221400	369	23748
9				13000280	766	48644
10				27116120	1586	99588

better or similar to the state-of-the-art (please see that S_i^{Lazy} always lies below S_i^{Greedy}). For the different applications and configurations the increase in scalability, measured as (S_i^{Greedy}/S_i^{Lazy}), of the `Lazy` scheduler in comparison to the state-of-the-art is on average 14% and maximally 30% higher based on Gustafson's Law. It can also be observed that the increase in the makespan estimation with respect to the base configuration becomes higher for configurations with higher workload and higher number of processors. This trend can be explained by observing the plot of S_i^{∞} that also increases with the increase in workload. In other words, the length of the longest path of the applications also increases when the input size increases.

VII. Related work

A parallel program is more complex to analyze than a sequential program due to the existence of an exponentially growing number of interleavings with the number of threads. Recently, architectures of time-predictable multicores have been proposed [14], [18]. In such architectures, the upper bound on accessing a shared hardware resource is bounded (predictable). Time-predictable architectures are increasingly receiving interest in analyzing timing behavior of parallel applications [17], [13], [15], [20]. Model checking is used in the work of Gustavsson et al. [7] by modeling spin-locks, private and shared caches to derive the makespan of small parallel programs. However, the approach in [7] suffers from state space explosion as the number of tasks increases.

The work by Rochange et al. [17] considers computing

makespan of a hard real-time parallel 3D multigrid solver running on a time-predictable MERASA multicore processor. Similar to our approach, [17] also considers dividing the code in parts that can execute in parallel. The main challenge addressed in [17], however, is to estimate an upper bound on the delay of synchronizations. Ozaktas et al. [13] also propose techniques to compute an upper bound on the stall time due to synchronizations.

The work in [15] proposes an approach to compute makespan of parallel programs where sequential tasks execute on different cores and they communicate via messages. The main idea in [15] is that the entire application is analyzed using a graph that connects the control-flow graphs of each task using edges used to model communication channels across threads. However, all of these works [17], [13], [15] assume that (i) the number of threads is no larger than the number of cores, and (ii) each thread is statically assigned to one core. Furthermore, [10] considers barrier-based synchronous execution. [19] study the use of OpenMP tasks for real-time applications. Next, [12] provides an overview of the available WCET analysis methods of parallel applications. There exists, however, to the best of our knowledge, no work that considers computing the makespan of a dynamically scheduled parallel application based on simulation of the underlying scheduling algorithm. The approach presented in this paper is a starting point for bridging the advances in real-time systems with the advances in modern task-based parallel programming models.

VIII. Conclusion

The paper addresses the problem of estimating the makespan of dynamically scheduled task-based parallel applications. A priority-assignment policy for the tasks of a parallel application is proposed. To determine the makespan of applications while avoiding timing anomalies, the design of the `Lazy` scheduler is presented and proved to be free from execution timing anomalies. Based on the WCET of each task, the estimation of the makespan by simulating the `Lazy` scheduler provides tighter bound (on average 9%) and better scalability (on average 14%) in comparison to the state-of-the-art technique for four parallel applications from the BOTS suite. To the best our knowledge, the `Lazy` scheduler is the first dynamic scheduler that is timing-anomaly free which offers tight estimation of the makespan and achieves good performance for task based parallel applications.

Acknowledgments

We thank the anonymous reviewers for their insightful comments on the earlier versions of the paper. We also thank all the members of MECCA project for their feedback. This research has been funded by the MECCA project under the ERC grant ERC-2013-AdG 340328-MECCA.

References

[1] E. Ayguadé et al. The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3), 2009.

[2] S. Baruah, V. Bonifaci, and A. Marchetti-Spaccamela. The global edf scheduling of systems of conditional sporadic dag tasks. In *Proceedings ECRTS*, pages 222–231. IEEE, 2015.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press Cambridge, 2001.

[4] A. Duran et al. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Parallel Processing, 2009. ICPP*, pages 124–131. IEEE, 2009.

[5] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[6] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[7] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson. Towards wcet analysis of multicore architectures using uppaal. In *Proceedings, WCET'10*, volume 15, 2010.

[8] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *Proc. of ECRTS*, 2014.

[9] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings RTSS, 1999.*, pages 12–21. IEEE, 1999.

[10] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho. Response-time analysis of synchronous parallel tasks in multiprocessor systems. In *Proceedings of RTNS*, page 3. ACM, 2014.

[11] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo. Response-time analysis of conditional dag tasks in multiprocessor systems. In *Proceedings, ECRTS*, pages 211–221. IEEE, 2015.

[12] V. Nélis, P. Meumeu Yomsi, and L. M. Pinho. Methodologies for the wcet analysis of parallel applications on many-core architectures. In *Proceedings DSD*, pages 748–755. IEEE, 2015.

[13] H. Ozaktas, C. Rochange, and P. Sainrat. Automatic wcet analysis of real-time parallel applications. In *Proceedings of WCET*, pages pp–11, 2013.

[14] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 57–68. ACM, 2009.

[15] D. Potop-Butucaru and I. Puaut. Integrated worst-case execution time estimation of multicore applications. In *Proceedings of WCET*, volume 30, 2013.

[16] J. Reineke et al. A definition and classification of timing anomalies. In *Proceedings of WCET*, volume 4, 2006.

[17] C. Rochange et al. Wcet analysis of a parallel 3d multigrid solver executed on the merasa multi-core. In *Proceedings of WCET*, volume 15, 2010.

[18] M. Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, 2009:2, 2009.

[19] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones. Timing characterization of openmp4 tasking model. In *Proceedings of CASES*, pages 157–166. IEEE Press, 2015.

[20] J. Wolf et al. Rtos support for parallel execution of hard real-time applications on the merasa multi-core processor. In *Proccedings of ISORC*, 2010.

Chapter 5

Paper II

Bounding the Execution Time of Parallel Applications on Unrelated Multiprocessors

Petros Voudouris, Per Stenström, Risat Pathan

Real-Time Systems journal, Springer (Under revision), 2021.

Real-Time Systems manuscript No.
(will be inserted by the editor)

Bounding the Execution Time of Parallel Applications on Unrelated Multiprocessors

Petros Voudouris · Per Stenström · Risat Pathan

Received: date / Accepted: date

Abstract Heterogeneous multiprocessors can offer high performance at low energy expenditures. However, to be able to use them in hard real-time systems, timing guarantees need to be provided, and the main challenge is to determine the worst-case schedule length (also known as makespan) of an application. Previous works that estimate the makespan focus mainly on the independent-task application model or the related multiprocessor model that limits the applicability of the makespan. On the other hand, the directed acyclic graph (DAG) application model and the unrelated multiprocessor model are general and can cover most of today's platforms and applications. In this work, we propose a simple work-conserving scheduling method of the tasks in a DAG and two new approaches to finding the makespan. A set of representative OpenMP task-based parallel applications from the BOTS benchmark suite and synthetic DAGs are used to evaluate the proposed method. Based on the empirical results, the proposed approach calculates the makespan close to the exhaustive method and with low pessimism compared to a lower bound of the actual makespan calculation.

Keywords Scheduling, heterogeneous, unrelated, DAG, work-conserving, makespan

1 Introduction

There is a continuously increasing demand for computational power in hard real-time systems, such as collision avoidance and mitigation function in automotive vehicles. Such a need for higher computing performance and energy efficiency has turned the focus both in academia and industry to heterogeneous multiprocessors (Esmaeilzadeh et al., 2011; Peter Greenhalgh, 2011; ARM, 2011). Heterogeneous multiprocessors comprise multiple computational

cores with different performance and functional characteristics. A real-time parallel application can exploit such parallel heterogeneous architectures to meet challenging performance and energy efficiency demands. However, one of the main challenges to using such an architecture in a hard real-time system is to ensure *time predictability*; one has to guarantee the timeliness of a real-time parallel application a heterogeneous platform by designing an effective scheduling algorithm and doing the offline schedulability analysis. This paper, for the first time, addresses the problem of determining the worst-case schedule length (also known as the *makespan*) of parallel application on *heterogeneous* multiprocessor platform and proposes a scheduling algorithm.

We consider a general model of the application and the processing platform, which makes the results of this paper applicable to a wide variety of applications and hardware platforms. A parallel application is modeled as a directed acyclic graph (DAG) where such a DAG has a collection of nodes, i.e., tasks and directed edges between nodes, i.e., dependencies among the tasks. The expressive power of a DAG enables us to model various applications like a collection of independent tasks (Baruah et al., 2015a) and synchronous parallel tasks (Lakshmanan et al., 2010).

We consider also a general system model – *unrelated heterogeneous multiprocessor platforms* that consist of different *processor types*. On an unrelated processing platform, a task/node τ^i of a DAG may execute at a different speed than another task τ^j on a processor of the same type[1]. The task-to-processor relationship in an unrelated heterogeneous platform governs how fast a particular task executes at run time. The unrelated heterogeneous multiprocessor model is one of the most general processor models that we consider in this paper (the homogeneous and related heterogeneous multiprocessor models are special cases of the unrelated multiprocessor model).

Related works on scheduling real-time systems that consider the unrelated multiprocessor model have mainly focused on independent tasks (Andersson and Raravi, 2014; Chwa et al., 2015; Andersson and Raravi, 2016; Baruah et al., 2019) with no dependencies and typed DAGs (Yang et al., 2016; Han et al., 2019). Earlier works that consider a DAG as the application model have focused mainly on the related multiprocessor model (Bender and Rabin, 2000; Jiang et al., 2017). Our research bridges the gap between the work on the related and the unrelated models by considering both a general application model (DAGs) and a general processor model.

Scheduling algorithms play the central role in guaranteeing time predictability, i.e., computing the makespan of parallel applications. Due to the specific speed relationship that a task of the DAG has with a particular processor type, one of the main challenges is to design an effective scheduling algorithm that can well exploit all the computational units of a heterogeneous parallel

[1] Unlike unrelated heterogeneous multiprocessors, related heterogeneous multiprocessors (also known as uniform multiprocessors (Baruah et al., 2015a)) have a specific speed for each processor type and all the tasks execute at that specific speed on any processor of that particular processor type. The homogeneous multiprocessor model has exactly one processor type, and all the tasks execute at the same speed on *all* processors.

architecture. A second challenge is to perform offline schedulability analysis by considering the execution of the tasks under the scheduling algorithm so that a safe and tight upper bound on the worst-case schedule length (makespan) can be computed. Such a makespan can be used, for example, to determine whether the deadline of an application will be met or not when the system is actually put in mission.

Many of the well-known schedulability analysis techniques for homogeneous multiprocessors cannot be trivially applied to heterogeneous multiprocessors (Gupta et al., 2012). One of the fundamental problems is the presence of *timing anomalies* (Graham, 1969). Note that a timing anomaly is already known to exist for the homogeneous multiprocessor model, which is a special case of the unrelated multiprocessor model (Voudouris et al., 2017; Pathan et al., 2018; Chen et al., 2019). Therefore, an example of a DAG — similar to that of (Voudouris et al., 2017) can also be constructed to demonstrate the presence of timing anomalies in the unrelated multiprocessor model. A method to avoid such anomalies for homogeneous multiprocessors is to preserve strictly, also at run-time, the order of *start time of the execution of the tasks* that was determined at analysis time (Voudouris et al., 2017; Pathan et al., 2018; Chen et al., 2019). Unfortunately, enforcing such an order of starting the tasks' execution is not enough to guarantee anomaly-freedom on unrelated machines because of the different speed relationships that each task has with each processor type.

This paper proposes a scheduling algorithm called the Greedy scheduler for unrelated HEterogeneous platform ($\mathcal{GHE}$) that can schedule the tasks of a DAG on an unrelated heterogeneous platform. One of the salient features of $\mathcal{GHE}$ is that it is *work-conserving* (a.k.a. greedy) (Graham, 1969; Brent, 1974; Blumofe and Leiserson, 1999; Melani et al., 2015; Jiang et al., 2017) meaning that it always dispatches an available task whenever there is an idle processor. The scheduler $\mathcal{GHE}$ is also very general in the sense that it does not assume any specific policy like fixed or dynamic priority-based scheduling used in the literature. Since many of the fixed- and dynamic-priority-based scheduling algorithms are also work-conserving, the analysis of this paper is also applicable for such schedulers. Another facet of $\mathcal{GHE}$ is that it allows the migration of a task to some other processor to execute it at a higher speed. It will be evident later that the fact that the scheduler is work-conserving and the migratory nature of $\mathcal{GHE}$ allows us to formally derive the makespan of a parallel application and prove its correctness.

A rigorous formal analysis is conducted in this paper to tackle and understand the complex relationships the tasks of a DAG have with the unrelated processors' types. Two different approaches – namely `Comb` and `Fast` – are proposed to determine in two different ways the makespan of a DAG executing on an unrelated heterogeneous multiprocessor under the $\mathcal{GHE}$ scheduler. The two approaches `Comb` and `Fast` mainly differ in terms of making the tradeoff between the computational complexity and tightness of the computed makespan. The first approach, `Comb`, is based on an exhaustive search by considering *all* the possible ways the tasks of a DAG may execute on different processors. On

the other hand, the second approach, `Fast`, is based on considering *one* pessimistic worst-case regarding how the tasks can execute on the processors. The `Comb` approach computes a tighter makespan in comparison to that of using the `Fast` approach, but `Comb` has exponential time complexity while `Fast` can find the makespan in polynomial time.

To evaluate the proposed approaches, `Fast` , and `Comb` , we use real-world parallel applications from the BOTS benchmark suite (Duran et al., 2002) as well as randomly generated synthetic DAGs. We also compare the proposed approaches to similar work in the literature for homogeneous (Graham, 1969), related (Jiang et al., 2017) multiprocessors, and typed DAGs (Han et al., 2019) to demonstrate how much we pay for using more generalized models of the application, hardware, and the scheduler with respect to that of state-of-the-art. One of the major findings we have from this empirical study is that the makespan computed using the efficient `Fast` approach is very close to that computed using the `Comb`, i.e., our analysis using the polynomial-time approach does not significantly compromise the tightness of the computed makespan. To this end, this paper makes the following contributions:

- This paper considers a general application model using DAGs, and a general hardware model for unrelated machines, to propose a general work-conserving scheduler $\mathcal{GHE}$. Consideration of such general models makes the results of this paper widely applicable to a variety of real-time systems.
- `Comb`: An exhaustive search-based approach `Comb` is proposed to find the makespan using a high computational complexity in order to find a tight makespan. This approach is suitable for applications that have tight deadlines.
- `Fast`: In order to reduce the computational complexity to find the makespan using the exhaustive approach of `Comb`, this paper proposes the polynomial-time approach `Fast` that can be used to find the makespan for large applications with a less tight makespan.
- The experimental evaluation presents empirical results for real-world applications based on the OpenMP applications from the BOTS benchmark suite (Duran et al., 2002), which shows the applicability of our approach to practical applications. Moreover, synthetic DAGs are used to show the sensitivity of our proposed approach to different real-world parameters. The degree of tightness that `Fast` sacrifices to find the makespan of the OpenMP applications in polynomial time is no more than 3% that of `Comb`, which shows that our proposed analysis for `Fast` does not introduce too much pessimism.

The rest of the paper is organized as follows: Initially, Sections 2–4 introduce the system model, the details of the proposed $\mathcal{GHE}$ scheduler, and necessary definitions for the makespan calculation. Next, Section 5 provides the details of our two proposed approaches to compute makespan. Then, Section 6 evaluates the time complexity of the proposed approaches. We then evaluate the proposed methods in Section 7 quantitatively. Section 8 compares our approach with related work that uses more specialized assumptions

regarding the platform and application models. Section 9 presents the related work before we conclude the paper in Section 10.

2 System Model

We consider an unrelated heterogeneous multiprocessor platform with a total of M processors with different types of processors. Each of the M processors belongs to exactly one of the processor types. The type of processor specifies the specialty or uniqueness of the processor. For example, the big.LITTLE multiprocessor chip from ARM has two different processor types with multiple processors that belong to each such processor type (Peter Greenhalgh, 2011; ARM, 2011). We assume that an unrelated platform can have from one up to M processor types.

A parallel application G is modeled as a directed acyclic graph (DAG) such that $G = (V, E)$, where $V = \{\tau^1, \ldots, \tau^N\}$ is a set of N nodes that designate tasks and $E \subseteq (V \times V)$ is a set of directed edges that designate dependencies among tasks. If $(\tau^p, \tau^q) \in E$, then τ^q can start its execution only after task τ^p completes. Tasks with no incoming and no outgoing edges are called source (denoted as τ^{src}) and sink (denoted as τ^{sink}), respectively. We assume that there is one source node and one sink node. If the application has multiple sources or sinks nodes, we add dummy nodes (i.e., nodes without execution time) to model the application.

A task is a sequential piece of code. Each task is characterized by a set of M worst case execution times (WCET) depending on the processor the task executes. Without loss of generality we index the processors from 1 to M. The WCET of task τ^i on the x^{th} processor is denoted by c^i_x. If a task cannot execute on the x^{th} processors, for example, due to an incompatible instruction set architecture, then $c^i_x = \infty$. Because the platform has a total of M processors, each task has M different WCETs $c^i_1, \ldots c^i_M$. If the x^{th} and y^{th} processors are of the same type, then $c^i_x = c^i_y$ for $i = 1, \ldots N$ where $1 \leq x \leq M$ and $1 \leq y \leq M$. In other words, each task τ^i has the same WCET on all the processors of the same type.

We define c^i_{min} as the minimum WCET of task τ^i for any of the processors in Eq. (1) as follows:

Definition 1 Minimum WCET of τ^i:

$$c^i_{min} := \min_{x=1}^{M} \{c^i_x\} \tag{1}$$

The *workload* of a task is the amount of computation that a task needs to complete when executing from the beginning to completion and is equal to c^i_{min} as is given in Eq. (1). The workload of a DAG G (denoted by $\mathcal{W}_1$) is the sum of the workloads of all the tasks in G and is given as follows:

Definition 2 Total workload of G:

$$\mathcal{W}_1 := \sum_{i=1}^{N} c^i_{min} \tag{2}$$

A source-to-sink path or simply a path γ in a DAG is a sequence of nodes $\gamma = (\tau^p, \tau^{p+1}, \ldots, \tau^{q-1}, \tau^q)$, where $(\tau^i, \tau^{i+1}) \in E$ such that $p \leq i < q$ and $\tau^p = \tau^{src}$ and $\tau^q = \tau^{sink}$. Let *paths* be the set of all the paths in a DAG G. The workload of a path γ is the sum of the workload of the nodes on that path and is given as follows:

$$\mathcal{W}(\gamma) := \sum_{\tau^i \in \gamma} c^i_{min} \tag{3}$$

The path with the largest workload among all the paths is called the longest or critical path (denoted using *cp*) and is given by Eq. (4):

$$cp = \arg \max_{\gamma \in paths} \mathcal{W}(\gamma) \tag{4}$$

The maximum workload of any path in G is given in Eq. (5):

Definition 3 The largest workload of any path in G:

$$\mathcal{W}_\infty = \mathcal{W}(cp) \tag{5}$$

While $\mathcal{W}_1$ represents the workload of the entire DAG, $\mathcal{W}_\infty$ is the maximum workload of any path of the DAG. The parameters $\mathcal{W}_1$ and $\mathcal{W}_\infty$ can be computed in polynomial time in the representation of the DAG and capture two important characteristics of the DAG that we will use to derive the makespan using our proposed approaches `Comb` and `Fast`. Since the definition of workload considers the minimum WCET of the nodes, no DAG can finish execution earlier than $\mathcal{W}_\infty$ (i.e., a lower bound on the makespan of G).

The workload of a path is constant because it is determined by the minimum WCET among the processors of the tasks that belong to the path. However, the duration that it will take to execute a path's workload can change from execution to execution at runtime. The WCET of the path's tasks can be larger than their minimum WCET because, during runtime, they may be mapped to slower processors than the processors that provide the minimum WCET. Consequently, the path with the largest workload is not necessarily the path with the longest time duration to complete its execution. This situation is illustrated in Fig. 1.

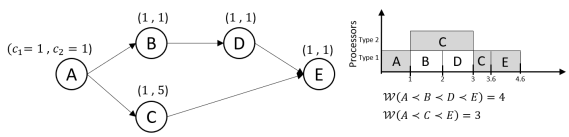


Fig. 1: Example of a path with the largest workload $A \prec B \prec D \prec E$ that it is not the path with the longest time duration.

The left-hand side of Fig. 1 shows a DAG. We use a platform with two processors of different processor types. Thus, each node has two different WCET. Using Eq. (3) that finds the workload of a path, the path $A \prec B \prec D \prec E$ has workload four while the path $A \prec C \prec E$ has workload three. At the right-hand side of the figure, we schedule the tasks based on their names' lexicographic order. From this example, we can see that path $A \prec C \prec E$ determines the schedule length, which has a smaller workload than $A \prec B \prec D \prec E$. Similar examples can be created, also for other than the lexicographic order of the tasks. To determine the makespan of a DAG, the key is to find the worst-case task-processor mapping (next section) that can occur during runtime for any execution ordering of DAG tasks. We will use the worst-case processor mapping together with the total workload and the critical path's workload to find the makespan of a single DAG.

To model the capabilities of the unrelated processors to execute the workload of a task, we define δ_x^i given in Eq. (6) the speed that task τ^i can execute on the x^{th} processor.

Definition 4 Speed of τ^i on the x^{th} processor for $x = 1, \dots M$:

$$\delta_x^i := \begin{cases} \frac{c_{min}^i}{c_x^i} & \text{if } c_x^i \neq \infty \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

If a task τ^i cannot execute on the x^{th} processor, we set the speed $\delta_x^i = 0$. Note that $0 \leq \delta_x^i \leq 1$ for any task τ^i. The smaller the WCET of τ^i on a particular processor, the larger the speed that task τ^i can execute on that processor is.

We define $\mathcal{O}_y^i$ as the y^{th} fastest speed that task τ^i can execute on some processor. For example, $\mathcal{O}_1^i$ for $y = 1$ specifies the fastest speed that task τ^i can execute (recall that the fastest speed is 1), $\mathcal{O}_3^i$ for $y = 3$ specifies the third highest speed that task τ^i can execute, and finally $\mathcal{O}_M^i$ for $y = M$ specifies the lowest speed that task τ^i can execute on some processor. It will be evident shortly that the design of $\mathcal{GHE}$ scheduler is such that it always prefers a relatively higher speed processor to execute a task. To that end, we specify the *preference for speed* of a task τ^i using a sequence $\mathcal{O}^i$ in Definition 5.

Definition 5 Let $\mathcal{O}^i$ be the sequence of a *non-increasing* order of speeds such that $\mathcal{O}^i =< \mathcal{O}_1^i, \mathcal{O}_2^i \dots \mathcal{O}_M^i >$ where $\mathcal{O}_y^i$ is the y^{th} fastest speed that task τ^i can execute on a processor of the platform for $y = 1, \dots M$.

In the next section, the scheduler will use the preference for speed ($\mathcal{O}^i$) to determine at which processor a task can execute. We will use $\mathcal{O}_y^i$ to specify the the minimum preference of speed, i.e., the maximum speed, at which the task τ^i can execute if all the processors that can execute task τ^i with higher speeds $\mathcal{O}_1^i, \mathcal{O}_2^i, \dots \mathcal{O}_{(y-1)}^i$ are busy. The $\mathcal{O}^i$ is a key component of our approach because it allows us to determine the preference of the processor for every task. Intuitively, in contrast to homogeneous and related multiprocessors in which all the tasks have the same view of the platform (same speeds), for unrelated multiprocessors, the $\mathcal{O}^i$ shows that every task views the platform differently because every task can have different speeds on the same processors.

3 Scheduler $\mathcal{GHE}$

We present in Section 3.1 the details of our proposed $\mathcal{GHE}$ scheduler. An important property of $\mathcal{GHE}$, called the Greediness Property, is stated in Lemma 1. Finally, we use an example in Section 3.2 to illustrate the working of the scheduler using the parameters of the system model.

3.1 Scheduler description

The $\mathcal{GHE}$ scheduler dispatches a new task awaiting execution in the ready queue when some other task finishes its execution (i.e., when some processor becomes idle). $\mathcal{GHE}$ is a work-conserving scheduler in the sense that it always dispatches a ready task if there is an idle processor. More precisely, the tasks are scheduled using $\mathcal{GHE}$ based on the next two steps: (i) Migration and (ii) Dispatching.

- **Step 1 - Migration**: If a processor becomes idle, the $\mathcal{GHE}$ scheduler first checks if the processor that becomes idle can execute some already executing tasks at a relatively higher speed. Without loss of generality, assume that τ^{mig} executes on the processor to which it has been migrated at its y^{th} fastest speed $\mathcal{O}_y^{mig}$. It is necessary for migrating τ^{mig} that no other executing task can execute at its k^{th} fastest speed for $k < y$ (please note that a lower index specifies a higher speed) on that idle processor.
- **Step 2 - Dispatching**: If there are tasks in the ready queue and there are idle processors, the $\mathcal{GHE}$ scheduler starts dispatching one-by-one new tasks awaiting execution in the ready queue on the fastest idle processor among all the idle processors.

The $\mathcal{GHE}$ scheduler is given in Algorithm 1. The scheduler is invoked each time some tasks finish their execution. The set of ready tasks and the indices of the processors that are idle are determined in variables `readyTasksSet` and `idleProcSet` (line 2–3), respectively. The set of tasks currently in execution is determined in variable `potenMigTasksSet` (line 4), and we consider these tasks for migration to an idle processor so that they can enjoy a higher speed. The set of indices of the busy processors executing the tasks in set `potenMigTasksSet` is determined in variable `busyProcSet` (line 5).

The while loop in lines 6-27 continuously checks if any of the currently executing tasks in set `potenMigTasksSet` can be migrated. If no such task can be migrated to any idle processor so that the task enjoys a higher speed, the while loop exits (line 24–26), and new tasks are dispatched using the second while loop in line 28–36.

The while loop in line 7 initializes the variable `anyMigration` to false. Line 8 initializes a set `noMigTasksSet` as an empty set that will be used to store the subset of the tasks of set `potenMigTasksSet` that are not selected for migration in the current iteration of the first while loop. The for loop in line 9–23 in each iteration considers a task τ^{mig} from set `potenMigTasksSet` for

migration from its current processor to an idle processor on which it would run relatively faster.

Line 10 determines the index `indexMigProc` of the processor such that task τ^{mig} is executing at its j^{th} highest speed on that processor with index `indexMigProc`. Line 12 determines the task τ^{find} among all the potential tasks for migration from set `potenMigTasksSet` that can be migrated to an idle processor with index `indexMigProc` such that task τ^{find} executes at the k^{th} highest speed and there is no other task from set `potenMigTasksSet` that can execute at h^{th} highest speed for some $h < k$. In other words, τ^{find} can execute on its most preferred processor in comparison to any other task in set `potenMigTasksSet`.

The condition in line 14 determines if the task τ^{mig} is the same as task τ^{find} and τ^{mig} can be executed at higher speed after migration, then τ^{mig} is migrated to the processor with index `indexMigProc` in line 15. The set of indices of the idle processors is updated in line 16–17 and the flag `anyMigration` is set to true to specify that migration has occurred during the current iteration of the while loop. If the condition in line 14 is false, then task τ^{mig} is not migrated in the current iteration of the while loop and stored in set `noMigTasksSet` in line 20 to consider for migration during the next iteration of the while loop. Regardless of whether migration occurs or not, task τ^{mig} is removed from set `potenMigTasksSet` and the for loop continues to consider another task from set `potenMigTasksSet` for migration. In other words, the flag `anyMigration` is set to true if one or more tasks from set `potenMigTasksSet` are selected for migration; otherwise, flag `anyMigration` will remain false.

When the for loop in line 9–23 completes, it is checked if migration occurred during the current iteration of the while loop or not based on the flag `anyMigration`. If the flag `anyMigration` is false, then the while loop is exited; otherwise, the while loop tries to migrate another task.

After the first while loop in line 6-27 completes, the second while loop in line 28-36 assigns the ready queue tasks to the idle processors. In line 29, an arbitrary task τ^{dis} from set `readyTasksSet` is selected, and it is assigned to the idle processor on which it would run the fastest.

In line 30, the index of the processor `indexNewProc` is searched such that `indexNewProc` $\in$ `idleProcSet` and task τ^{dis} executes on its k^{th} fastest processor and cannot execute faster on any of the processors in set `idleProcSet`. Finally, we update the set of idle processors, and we remove the task from the ready queue. One by one, a new task from set `readyTasksSet` is dispatched to an idle processor as long as there are new tasks in the ready queue and there is at least one idle processor.

Next, we present a property, called the *Greediness Property*, of the scheduler in Lemma 1. A scheduling point is a time instant when the scheduler needs to make some new decision. Such a trivial scheduling point is at time zero. In addition, there is a scheduling point every time some task finishes its execution. We denote a time interval $[a,b]$ a *stable time interval* such that there is no scheduling point inside the interval except at the endpoints in $[a,b]$.

Lemma 1 proves the worst-case speeds that the tasks can execute during any stable time interval $[a,b]$ when scheduled using the $\mathcal{GHE}$ scheduler.

Algorithm 1: The $\mathcal{GHE}$ Scheduler

```
1  if some task completes execution then
2      readyTasksSet = Set of task that are in the ready queue
3      idleProcSet = Set of indices of processors that are idle
4      potenMigTasksSet = Set of unfinished tasks in execution
5      busyProcSet = Set of indices of processors that is currently
         executing some unfinished task
6      while true do
7          anyMigration = false
8          noMigTasksSet = ∅
9          for each τ^mig ∈ potenMigTasksSet do
10             (j, indexCurProc) = τ^mig is currently executing at its j^th
                 fastest speed on processor with index indexCurProc
11
12             (k, indexMigProc, τ^find) = Find the smallest index k and
                 task τ^find from set potenMigTasksSet such that (i) τ^find
                 can execute on its k^th highest processor that has index
                 indexMigProc where indexMigProc ∈ idleProcSet and
                 (ii) no other task in potenMigTasksSet can execute on a
                 processor in idleProcSet at its h^th highest speed where
                 h < k.
13
14             if k < j and τ^mig = τ^find then
15                 Migrate task τ^mig from indexCurProc to indexMigProc
16                 idleProcSet = idleProcSet − {indexMigProc}
17                 idleProcSet = idleProcSet ∪ {indexCurProc}
18                 anyMigration = true;
19             else
20                 noMigTasksSet = noMigTasksSet ∪ {τ^mig}
21             end
22             potenMigTasksSet = potenMigTasksSet − {τ^mig}
23         end
24         if anyMigration == false then
25             break and exit the while loop in line 6–28
26         end
27     end
28     while idleProcSet ≠ ∅ and readyTasksSet ≠ ∅ do
29         τ^dis = any task from set readyTasksSet
30
31         indexNewProc = Find the index of the processor such that
             indexNewProc ∈ idleProcSet on which task τ^dis would
             execute fastest among all other processors in set idleProcSet
32
33         Dispatch task τ^dis to indexNewProc
34         idleProcSet = idleProcSet − {indexNewProc}
35         readyTasksSet = readyTasksSet − {τ^dis}
36     end
37 end
```

Lemma 1 ***Greediness Property****: If there are a total of p processors busy executing some tasks during any stable time interval under the $\mathcal{GHE}$ scheduler, then there is some task executing at least at its k^{th} speed for $k = 1, 2, \ldots p$ and $1 \leq p \leq M$.*

Proof Let k, where $0 \leq k \leq M$, be the set of tasks that continues execution from one stable time interval to the immediately next stable time interval. Also, assume that for some n, where $0 \leq n \leq M-k$, there are n tasks that are newly scheduled at the beginning of the next stable time interval. Let $k+n = p$, where $0 \leq p \leq M$, be all tasks that we need to consider for execution in the new stable time interval. Let $\mathcal{O}_x^*$ denote the x^{th} highest speed of some task.

We will prove this lemma considering two cases: (1) all the processors are idle, or (2) some processors are busy (i.e., some tasks from previous stable time interval continue their execution).

Case (1) - All processors are idle: If M processors are idle (i.e., $k = 0$, $n = p$), then the first task that we select can be scheduled to its fastest processor that has speed one ($\mathcal{O}_1^*$) because all the processors are available. Next, the second task that we select, in the worst-case, is scheduled on its second-fastest processor ($\mathcal{O}_2^*$) because the first task may occupy the processor that provides to the second task a faster speed. Finally, the p^{th} task (τ^p) in the worst-case is dispatched with speed $\mathcal{O}_p^p$. Because all the $p-1$ processors that provide higher speed ($\mathcal{O}_1^p$, $\mathcal{O}_2^p$, $\ldots$, $\mathcal{O}_{p-1}^p$) for τ^p may be occupied by other tasks. So for this stable time interval the p tasks are executing with $\mathcal{O}_x^*$, $1 \leq x \leq p$, respectively.

Case (2) - Some tasks are still in execution: We separate two sub-cases: In sub-case (2.a), the tasks that are still in execution do not migrate, and in sub-case (2.b), some of the tasks that are in execution would migrate.

Sub-case (2.a) - No migration: In this sub-case, only the new n tasks are scheduled on the idle processors while the k already-executing tasks continue executing on the processor on which they were executed in the previous stable interval. The first new task is going to execute at least with speed $\mathcal{O}_{k+1}^*$ because, in the worst-case, the k faster processors are occupied. Similarly, the remaining tasks from the n scheduled tasks are going to execute with speeds $\mathcal{O}_{k+2}^*$, $\ldots$, $\mathcal{O}_{k+n}^*$. Therefore, the $k+n = p$ tasks are executing with speeds $\mathcal{O}_x^*$, $1 \leq x \leq p$ in this stable time interval.

Sub-case (2.b) - Migration: Let τ^i complete its execution at time a which is at the beginning of the stable time interval [a,b]. Let τ^j be among the k tasks that still continue executing during the stable time interval [a,b]. Please recall that $\mathcal{GHE}$ first selects for migration the task that can enjoy its most preferred processor compared to other tasks. Let τ^j, if it migrates, has the most preferred processor among the k tasks that are still in execution. In the worst-case, τ^i was executing before its completion on a processor that is also for τ^j a faster processor. So τ^j can migrate to a processor that has at least $\mathcal{O}_k^j$ speed because there are k tasks that can occupy the faster processors for τ^j. By following the $\mathcal{GHE}$ scheduler, the speeds at which tasks would start executing from the beginning of the stable interval in the worst case are $\mathcal{O}_1^*$,

$\mathcal{O}_2^*, \ldots, \mathcal{O}_k^*$. The n tasks that we need to dispatch will have the speeds[2] in the worst-case $\mathcal{O}_{k+1}^*$, $\mathcal{O}_{k+2}^*$, $\ldots$, $\mathcal{O}_{k+n}^*$ (follows directly from the same argument as sub-case (2.a)). Therefore, the $k + n = p$ tasks are executing with speeds $\mathcal{O}_x^*$, $1 \leq x \leq p$ in this stable time interval.

The main idea of the greediness property is that if $x - 1$ processors are busy, then in the worst-case a task executes with its x^{th} fastest speed. The scheduler selects an arbitrary task to dispatch. We prove the scheduler's greediness without assuming any priority of the tasks. Because greediness property is oblivious to the priorities of the tasks, it holds for any priority assignment that would allow us in the next section to find a makespan computation that also holds for any priority assignment of the tasks. Finding a priority assignment that would lead to a shorter makespan is an exciting and challenging problem. However, we do not address it in this paper. In addition, we can preserve the greediness property if the scheduler is extended with preemption capability by doing the preemption before we do migration and dispatch steps because the greediness property holds for any priority assignment. Preemptive scheduling may improve the makespan, but we need to consider the preemption cost and a larger number of migrations, as we explain next.

A note on migration. We assume that the cost of migrations is already included in the WCET of the task. The total cost of migration depends on the *total number* of migrations and the cost of *each* migration. Initially, in the worst-case based on the $\mathcal{GHE}$, the number of migrations for a task is bounded by $(M - 1)$ because the task can migrate from its slowest processor to its fastest processor by migrating at most $(M - 1)$ times. In the worst-case, a task needs to wait for $(M - 1)$ other tasks to migrate before it can migrate, in case that all the tasks that are in execution also need to migrate. Thus, in the worst case for each task we need $(M-1)\cdot(M-1)$ migrations to consider.

A note on preemption. In case the scheduler is preemptive, which is equivalent to temporarily removing the tasks that are already in execution and consider them all for dispatching based on some priority order, the number of migrations that we need to consider for each task is higher compare to non-preemptive scheduling. Because when a task continues its execution after being preempted, it may be scheduled to a slower processor compared to the processor that was executing before it was preempted. As a result for preemptive scheduling the maximum number of migrations is $(N-1)\cdot(M-1)\cdot(M-1)$. Since we can bound the maximum number of migrations, the proposed scheduler is suitable for worst-case timing analysis. Finding the cost of each migration for heterogeneous multiprocessors is a challenging problem that we do not address in this paper. The migration cost is platform-dependent, and the platform architectural characteristics are known during the WCET analysis. We assume that each migration's cost can be computed and included within the task's WCET.

[2] The asterisk is used as a wildcard task executing at its preferred processor.

3.2 An Example

This section presents an example of the application, platform, and scheduler using the parameters that we have defined in earlier sections. We are also going to refer to this example in later sections of this paper.

Figure 2 shows an application that we model as a DAG with six nodes A–F and seven dependencies. We assume an unrelated multiprocessor platform with two processors where each processor belongs to one unique type. The set of the WCETs of the tasks are shown in Table 1. Note that $c_i^1 \neq c_i^2$ for some (in this case for all) tasks, which implies that the types of the two processors are different. There are two types of processors denoted as type 1 and type 2.

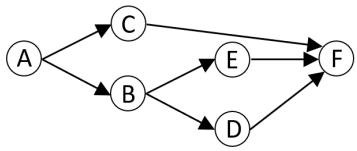


Fig. 2: The DAG of an application with six nodes and seven dependencies.

	c_i^1	c_i^2
A	1	2
B	1	10
C	10	1
D	2	1
E	1	2
F	1	2

Table 1: The WCET of the nodes in Figure 2 for two processors.

In Table 2, we calculate the total workload and the workload of the critical path for three cases. The column labeled as "Both type 1" is used to specify a homogeneous multiprocessor platform with two processors where both processors are of type 1. Similarly, the column labeled as "Both type 2" is used to specify a homogeneous multiprocessor platform with two processors where both processors are of type 2. Finally, the column labeled as "Unrelated: one type 1 and one type 2" is used to specify a heterogeneous multiprocessor platform with two processors where one processor is type 1, and the other is type 2.

With Eq. (2) and Eq. (5), we calculate the total workload $\mathcal{W}_1$ in the second row and the workload of the critical path $\mathcal{W}_\infty$ in the third row for all the three cases. Please note that Eq. (2) and Eq. (5) can also be applied to homogeneous multiprocessors as there is only one WCET for homogeneous multiprocessors that is equal to the minimum WCET. Eq. (7) is widely used in previous works (Graham, 1969; Brent, 1974; Blumofe and Leiserson, 1999; Melani et al., 2015) to compute the makespan for homogeneous multiprocessors of tasks that are scheduled by a work-conserving scheduler.

$$T_{M(1)} = \frac{\mathcal{W}_1 + (M-1) \cdot \frac{\mathcal{W}_\infty}{1}}{M} \tag{7}$$

For the platform with two homogeneous type 1 processors, we get a makespan equal to 13.5, and for the platform with two homogeneous type 2 processors, we get a makespan equal to 17.

	Both type 1	Both type 2	Unrelated: one type 1 and one type 2
$\mathcal{W}_1$	16	18	6
$\mathcal{W}_\infty$	11	16	4
Makespan	13.5	17	to be determined

Table 2: Total workload, workload of the longest and the makespan of the DAG are presented in Figure 2 for two homogeneous platforms and one unrelated platform. In Section 5, we will discover the makespan with the unrelated multiprocessor platform is equal to 7.28.

However, Eq. (7) cannot be trivially applied to heterogeneous multiprocessors because it does not take into account the heterogeneity of the processors. The related multiprocessors (Jiang et al., 2017) take into account the heterogeneity of the processors but assume that all the tasks can benefit equally from the architectural characteristics of the available processors, which is not realistic because they can benefit differently from the different processor types.

Before we determine the makespan for unrelated multiprocessors we first analyze the simulation of the execution of the DAG at Figure 2 for three different scenarios.

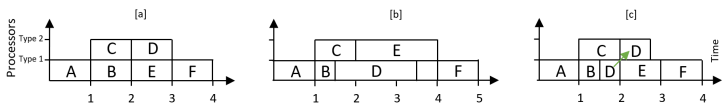


Fig. 3: Case (a) presents the simulation of the execution on the unrelated platform if all the tasks execute for their WCET, case (b) presents the simulation if τ^B completed its execution earlier without migration, and case (c) presents the simulation with task migration.

Figure 3.a presents the simulations of the execution if all the tasks execute for their WCET on the unrelated platform based on our proposed $\mathcal{GHE}$ scheduler. It can be seen that the schedule length is 4 time units. So the unrelated multiprocessor platform can execute the tasks by taking advantage of the heterogeneity of the platform and can benefit from the different architectural characteristics of the processors.

In Figure 3.b, τ^B completes its execution earlier than its WCET. Since the scheduler is work-conserving, task τ^D is dispatched to the processor of type 1 with speed 0.5. Similarly, τ^E is dispatched to the processor of type 2 that also has the speed 0.5, and schedule length is 5. This schedule does not allow migration.

In Figure 3.c, we assume that the scheduler migrates a task to the other processor to enjoy faster speed. So, after the completion of τ^C, task τ^D migrates to a type 2 processor to continue at a higher speed. Next, τ^E is dispatched to type 1 to execute with speed one, and the schedule length is 4.

The simulations of the execution of Figures 3.a and 3.c show the expected behavior of $\mathcal{GHE}$. However, Figure 3.b does *not* represent the expected behavior of $\mathcal{GHE}$ because between time 2 to 4 task τ^D and τ^E are both executing with their second-highest speed that violates the greediness property of Lemma (1). The simulation of the execution assuming that all tasks execute for their WCET cannot be used to calculate the makespan because of timing anomalies. We are trying to provide a safe upper bound of the worst-case schedule length (makespan).

4 Formal tools to compute the makespan

The schedulability analysis of DAGs on an unrelated multiprocessor platform in contrast to homogeneous and related platforms cannot be done without taking into account the application. We can analyze a homogeneous platform for any DAG by knowing only the number of processors. For a related platform that has processors of different speeds, based on (Funk et al., 2001; Jiang et al., 2017) we need two parameters. First, the capacity of the platform is the sum of the processors' speeds and shows the rate that the workload of the application is executed for a given number of processors. Second, the uniformity intuitively shows how much the processors' speed differs compared to a homogeneous platform with the same number of processors. The capacity and uniformity of a platform are fixed for any DAG for which we want to estimate the makespan. However, for an unrelated platform, the WCET of the tasks depends on the task-processor mapping; the platform characteristics can be different for every DAG.

Our approach to characterize a platform is to extend the concept of capacity and uniformity from (Funk et al., 2001; Jiang et al., 2017) by taking into account the scheduler and the speeds that are derived by the task-processor mappings. Section 4.1 presents preliminary definitions and the motivation behind defining the minimum capacity and heterogeneity that we formally present in Sections 4.2 and 4.3, respectively.

4.1 Motivation and preliminary definitions

Computing the actual makespan of a DAG executed on unrelated multiprocessors is intractable (Garey and Johnson, 2002), so we focus on computing an upper bound on the makespan. This section presents the motivation for formal analysis tools that we will use in the next section to find a safe upper bound on the makespan.

Initially, let us focus on the execution of a single task τ^i and visualize its execution as rectangles. The vertical side is the speed at which the task is

executing, and the horizontal side its execution time. The rectangle's surface area shows the workload of the task given by Eq. (1).

Let τ^i have a workload equal to two. Figure 4 shows an abstract view of the execution of τ^i for three scenarios. First, (a) shows the cases that τ^i is mapped to a processor that provides to τ^i speed one, so τ^i completes its execution after two time units. By computing the area of the rectangle, we find the workload of τ^i which is two. At (b), the τ^i is mapped to a processor that offers speed 0.5 to τ^i and as a result, the execution time is four, which is greater compared to (a). Again the workload is two by computing the area of the rectangle. At (c) τ^i initially is mapped to a processor that offers speed 0.5 for two time units, and then it migrates to a processor that provides speed one for one time unit. The execution time for τ^i for the (c) case is three time units, but again the sum of the area of the rectangles is two, which is equal to the workload of the τ^i.

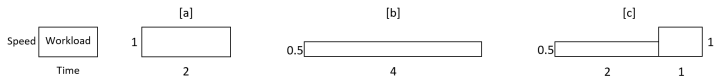


Fig. 4: Visualization of the execution of a task τ^i with workload equal to two, when (a) is mapped to a processor with speed 1, (b) with speed 0.5 and (c) initially with speed 0.5 and then it migrates to a processor with speed 1. Whatever is the task-processor mapping the sum of the area of rectangles for all the cases is constant and equal to the workload.

This example illustrates that the task-processor mapping can lead to different execution times. However, we observe that the workload of the task remains constant regardless of the task-processor mapping. Based on this observation, we will inflate the DAG's workload to bound all the schedule lengths that we can get for all the possible task-processor mappings. The inflation will introduce pessimism to the schedule length of the DAG that captures all the possible task-processor mappings in order to find a safe estimation of the makespan.

Next, let us try to visualize with Figure 5 the schedule of a DAG on an unrelated multiprocessor platform as a two-dimensional area. Let the $Capacity$ be the sum of the speeds based on the task-processor mapping of the tasks that are in execution. The vertical axis shows the capacity. The horizontal axis is the time duration the application takes to complete its execution. Let B be the area where processors are busy executing some workload, and $\bar{B}$ be the area that the processors are idle. So the makespan is given by:

$$Makespan = \frac{B + \bar{B}}{Capacity} \tag{8}$$

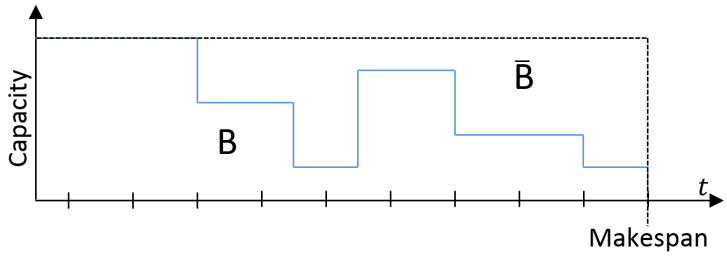


Fig. 5: Visualization of the B and $\bar{B}$.

To compute the makespan for unrelated multiprocessors, we need to find the terms of Eq. (8) that maximize the numerator and minimize the denominator.

First, regardless of the execution time of the application, the total workload ($\mathcal{W}_1$) is equal to the area in B. At run-time, a node can run with a speed that is lower than one (smaller vertical-value), which would have a proportionately longer duration (larger horizontal value), as we explained in Figure 4. So, the area of B, which is composed of the workload of all the tasks, remains constant and is equal to the total workload of the DAG.

We can have different capacities during different stable time intervals because the speeds depend on the task-processor mapping. The largest possible capacity of the platform is M when all the processors execute tasks with speed one. The smallest value is one, which is the case when one task is executing, and because the scheduler has the property of Lemma (1), it will schedule or migrate a task to its fastest (speed equal to one) available processor. To determine the capacity that would lead to the estimation of the makespan, we define as $\tilde{S_m}$ the minimum capacity among all the task-processor mappings that we can have based on scheduling decisions of $\mathcal{GHE}$ for $m \leq M$ processors.

To determine $\bar{B}$, we need the duration (horizontal value) that there are idle processors and the capacity of the unused processors (vertical value). Because we assume a work-conserving scheduler, we know that if there are idle processors and the application is not finished, there are still tasks that we need to execute, but they are restricted by their dependencies. Let $\mathcal{W}_\gamma$ denote the workload of an arbitrary path of the application. The workload of any path in the DAG can be at most the workload of the critical path. The critical path length is different depending on the task-processor mapping of the tasks that belong to the critical path. The shortest length of the critical path is $\mathcal{W}_\infty$, which is the case if all task-processor mappings have speed one. Let $\tilde{\mathcal{O}}$ denote the speed that leads to the worst-case (i.e., maximum) unused capacity for all the tasks. Thus, the $\frac{\mathcal{W}_\infty}{\tilde{\mathcal{O}}}$ shows the length of the critical path. Let $\tilde{idle}$ denote the unused capacity, and by replacing the terms to Eq. (8), we get:

$$T_M \leq \frac{\mathcal{W}_1 + i\tilde{dl}e \cdot \frac{\mathcal{W}_\gamma}{\tilde{\mathcal{O}}}}{\tilde{S_M}}$$

$$\implies T_M \leq \frac{\mathcal{W}_1 + i\tilde{dl}e \cdot \frac{\mathcal{W}_\infty}{\tilde{\mathcal{O}}}}{\tilde{S_M}}$$

$$T_M \leq \frac{\mathcal{W}_1 + \frac{i\tilde{dl}e}{\tilde{\mathcal{O}}} \cdot \mathcal{W}_\infty}{\tilde{S_M}}$$

It can be seen from Eq. (7) that the values of $\tilde{\mathcal{O}}$ and $i\tilde{dl}e$ for a homogeneous multiprocessor platform are one and $(M-1)$, respectively. However, the values of $\tilde{\mathcal{O}}$ and $i\tilde{dl}e$ are unknown for unrelated multiprocessors. Because the scheduler is work-conserving $i\tilde{dl}e$ depends on $\tilde{\mathcal{O}}$. We combine these two parameters, and we call this expression heterogeneity. To find the maximum heterogeneity, we need to search among the different task-processor mappings that are determined by the $\mathcal{GHE}$ scheduler. To enumerate all possible task-processor mappings, we introduce:

Definition 6 Let π be one permutation of p tasks selected from N tasks of the application. The set of all the permutations of size p selected from N different tasks is denoted by σ_p. The total number of permutations of size p selected from N tasks is $\frac{N!}{(N-p)!}$.

In Section 4.2, we define two ways to calculate the minimum capacity that takes into account the processor preference order of every task determined by the scheduler. In Section 4.3, we define heterogeneity among all the tasks and all the processors throughout the execution.

4.2 Minimum capacity

For any stable time interval (based on Lemma (1)), if $(x-1)$ processors are busy, then a task is executed at least on its x^{th} faster processor. Let $\mathcal{O}_k^{\pi(k)}$ denote the k^{th} speed of the k^{th} task in permutation π. During a stable time interval when x processors are busy, we define in Eq. (9) the minimum capacity of the platform that tasks in permutation $\pi \in \sigma_x$ can execute with:

Definition 7 Capacity of x processors for permutation π:

$$S_x^\pi := \sum_{k=1}^{x} \mathcal{O}_k^{\pi(k)} \tag{9}$$

The minimum capacity over all possible permutations $\pi \in \sigma_M$, denoted by S_M, is given by:

Definition 8 Minimum capacity of the platform among all permutations:

$$S_M := \min_{\pi \in \sigma_M} \{S_M^\pi\} \tag{10}$$

The time complexity to evaluate Eq. (10) is exponential because we need to search through all the permutations. To avoid the high complexity, we trade off precision, and we define in Eq. (11) an alternative approach to calculating the minimum capacity for $1 \leq m \leq M$ processors. Instead of searching among all the permutations, we search among all the tasks to find the minimum capacity. This approach is always more or equally pessimistic compared to Eq. (10), and the proof is presented in Appendix A.1.

Definition 9 Minimum capacity of the platform among all tasks and processors:

$$S_m^{'} := \sum_{x=1}^{m} \min_{i=1}^{N} \{\mathcal{O}_x^i\} \tag{11}$$

While the capacity of the hardware platform for homogeneous and related machine models does not change from one task to another, the capacity for unrelated machines changes from one DAG to another, the definitions of capacity in Eq. (10) and Eq. (11) are measures of the unrelated platform concerning the tasks, and they will be used in deriving the makespan of a DAG. Intuitively, the capacity shows the minimum rate that the workload of a DAG is executed for a given number of processors of the unrelated platform.

4.3 Heterogeneity

In this subsection, we use the notion of heterogeneity to capture how much capacity of the platform could be wasted (recall the area of unused capacity), which will be used to find the makespan. For a permutation π, based on Lemma (1), if $(x-1)$ processors are busy, a task in the worst-case will execute with speed $\mathcal{O}_x^{\pi(x)}$. Based on the $\mathcal{GHE}$, the unused capacity is given by S_M^π -S_x^π that depends on the speed $\mathcal{O}_x^{\pi(x)}$. To find the maximum unused-capacity area, for a permutation π we define with Eq. (12) the heterogeneity. It finds for a specific permutation the maximum heterogeneity among the different processors.

Definition 10 Heterogeneity for permutation π:

$$\lambda^\pi := \max_{x=1}^{M} \{\frac{S_M^\pi - S_x^\pi}{\mathcal{O}_x^{\pi(x)}}\} \text{ where, } \mathcal{O}_x^{\pi(x)} \neq 0 \tag{12}$$

To identify the maximum permutation-based heterogeneity (λ) we define Eq. (13), where we search among all the permutations.

Definition 11 Maximum heterogeneity among all permutations:

$$\lambda := \max_{\pi \in \sigma_M} \{\lambda^\pi\} \tag{13}$$

Similarly, with the permutation-based capacity, from Eq. (10), the calculation of the maximum heterogeneity, from Eq. (13), has exponential time complexity because we need to search through all of the permutations. To avoid the high complexity, we define heterogeneity by searching through all of the tasks and all the processors instead of searching among all the permutations. If the workload on the critical path ($\mathcal{W}_\infty$) is executed with speed $\mathcal{O}_x^i$ then the unused capacity is given by Eq. (14), where $\overline{\mathcal{O}}_x^i$ is the subset of $\mathcal{O}^i$, whose speed is lower than the x^{th} fastest processors for task τ^i.

Definition 12 Unused capacity if τ^i is scheduled on processor x:

$$idle_x^i := \sum_{y \in \overline{\mathcal{O}}_x^i} \max_{j=1}^{N} \{\mathcal{O}_y^j\} \tag{14}$$

To find the heterogeneity ($\lambda^{'}$), we combine Eq. (14) with its corresponding busy speed $\mathcal{O}_x^i$, and we maximize among the different tasks and different processors. The second version of heterogeneity is always more or equally pessimistic compared to Eq. (15), and we present the proof in Appendix (A.2).

Definition 13 Maximum heterogeneity among all tasks and processors:

$$\lambda^{'} := \max_{i=1}^{N} \{\max_{x=1}^{M} \{\frac{idle_x^i}{\mathcal{O}_x^i}\}\} \text{ where, } \mathcal{O}_x^i \neq 0 \tag{15}$$

For unrelated multiprocessors, it is not clear which path of the DAG would lead to the worst-case schedule length. However, in both approaches, to identify the heterogeneity, **all** the tasks and not just the tasks that belong to the critical path are considered. So the main idea is to combine heterogeneity, which takes into account all the possible tasks-processor mappings, with the critical path that has the largest workload among all the paths (Eq.(5)) to identify the maximum unused capacity area. Intuitively, the heterogeneity shows if the unrelated platform is appropriate for a DAG concerning an "ideal platform" with the same number of processors that all tasks will execute for their minimum WCET among the heterogeneous processors.

Figure 6 presents all the permutations for the application that was presented in Figure 2. For every permutation, we calculate the permutation-based capacity and the heterogeneity based on Eq. (9) and Eq. (12), respectively. The minimum permutation-based capacity given by Eq. (10) is 1.1 and the maximum permutation-based heterogeneity given by Eq. (13) is 0.5. To avoid the high time complexity of enumerating all the permutations, to find the capacity and the heterogeneity, we apply Eq. (11) and Eq. (15) to find the capacity and the heterogeneity in polynomial time complexity, and we also get 1.1 and 0.5, respectively. Now that we have all the parameters ready, we can move to the proof of the calculation of the makespan (next section) that would allow us to put all concepts together to find the makespan for unrelated multiprocessors.

π	A	B	C	D	E	F	• • •	A	B	C	D	E	F
	A	A	A	A	A	A		F	F	F	F	F	F
S_M^π	1.5	1.1	1.1	1.5	1.5	1.5		1.5	1.1	1.1	1.5	1.5	1.5
λ^π	0.5	0.5	0.5	0.5	0.5	0.5		0.5	0.5	0.5	0.5	0.5	0.5

Fig. 6: Permutations and the calculation of the permutation-based capacity and heterogeneity for the example given in Figure 2.

5 Makespan calculation

This section presents two approaches to calculate the makespan of a DAG on unrelated multiprocessor platforms. Initially, Section 5.1 provides the proof sketch of the proposed approaches. The first issue (Section 5.2) for both approaches is how to determine a formally proven upper bound on the execution time of the DAG (a requirement on safety). A makespan computation must never underestimate the length of the schedule to ensure that the real-time constraints are satisfied. Lemma (2) presents an exhaustive search-based combinatorial approach to calculate such an upper bound on the makespan. A safe upper bound on the makespan is determined by identifying all the permutations of possible task-processor mappings. This approach has exponential time complexity but provides a tight makespan estimation (as will be evident later in our experiments). As a result, such an approach can be used only for a small number of processors and tasks.

The second issue (Section 5.3) is to provide a tight estimation of the makespan that avoids exhaustive approaches and can be calculated efficiently. Theorem (1) proposes a polynomial time complexity makespan calculation. Based on Lemma (3) and Lemma (4), it is proven that the makespan calculated by Theorem (1) is always greater or equal than the (exhaustive) makespan given by Lemma (2) but still quite tight. As a result, the second approach is shown to be always a safe estimation of the makespan.

5.1 Overview

To find a safe and tight bound on the makespan, we partition an arbitrary schedule across different time intervals such that the number of busy processors in each such interval is constant. We use an exhaustive approach, and for every interval, we search through all of the permutations of task-processor mappings. To identify for every interval the permutation that leads to the worst-case schedule length, we adapt for the unrelated model (minimum capacity and heterogeneity) two well-known parameters used earlier in the context of related multiprocessors (Funk et al., 2001; Jiang et al., 2017). These two parameters that try to maximize the schedule length are combined with the total workload and the workload of the critical path of the DAG to compute the makespan.

The initial, exhaustive approach has exponential time complexity since all the permutations need to be searched. As a result, it is applicable to a restricted number of tasks and processors. To address this limitation, we propose a polynomial time-complexity version of the heterogeneity and capacity that is independent of the permutations and formally proven to be always more pessimistic compared to the permutation-based version of the parameters.

5.2 Exhaustive search makespan

In this section, we present our first result in Lemma (2) that can be used to compute the makespan. The final formula of the makespan uses the total workload and the workload of the longest path presented in Section 2 and the permutation-based minimum capacity and heterogeneity introduced in Sections 4.2 and 4.3, respectively.

Lemma 2 *Exhaustive makespan (`Comb`): The makespan of a DAG executed on an unrelated multiprocessor platform is given by:*

$$T_M^{\texttt{Comb}} \leq \frac{\mathcal{W}_1 + \lambda \cdot \mathcal{W}_\infty}{S_M} \tag{16}$$

Proof Let B_p denote the sum of the lengths of the time intervals where exactly p processors are busy. Because $\mathcal{GHE}$ is work-conserving, we know that there will always be at least one processor busy during the execution of the application. So for the makespan $T_M^{\texttt{Comb}}$, we have:

$$T_M^{\texttt{Comb}} = \sum_{p=1}^{M} B_p \tag{17}$$

For an application that is executed on an unrelated multiprocessor platform, the different task-to-processor mappings can lead to different schedule lengths. To describe all the task-processor mappings when exactly p processors are busy, we introduce B_p^π, which denotes the time interval when exactly p processors are busy by the tasks of permutation π of size p. Since we can have different permutations that occupy p processors, it holds that:

$$B_p = \sum_{\pi \in \sigma_p} B_p^\pi \tag{18}$$

where σ_p is the set of all permutations of size p. If during run-time a permutation π' does not appear in the schedule, then it holds that $B_p^{\pi'} = 0$.

Consider an arbitrary schedule and an arbitrary busy interval B_p^π. Based on Lemma (1), during B_p^π, the p^{th} task belonging to π in the worst-case will be executed with speed $\mathcal{O}_p^{\pi(p)}$ where $\mathcal{O}_p^{\pi(p)} \neq 0$. Let $\mathcal{W}^{p,\pi}(\gamma)$ denote the total amount of workload completed at speed $\mathcal{O}_p^{\pi(p)}$ from task $\tau^{\pi(p)}$ that belongs to an arbitrary path γ and at the p^{th} position of permutation π and we have:

$$B_p^{\pi} \cdot \mathcal{O}_p^{\pi(p)} \leq \mathcal{W}^{p,\pi}(\gamma)$$

We break up the workload of the tasks that belong to the critical path $\mathcal{W}_\infty$ into fragments that depend on the task-processor mappings; that is, fragments that depend on permutations π of size p denoted by $\mathcal{W}_\infty^{p,\pi}$. If for a permutation π, it holds that $B_p^{\pi} = 0$, then it also holds that $\mathcal{W}_\infty^{p,\pi} = 0$ because this permutation did not appear in the schedule, so it does not have any workload. With $\mathcal{W}_\infty^{p}$, we denote the workload of the critical path of all permutations executed by the same number of processors. To collect the workload from all the permutations, we define:

$$\mathcal{W}_\infty \geq \sum_{p=1}^{M} \mathcal{W}_\infty^{p} = \sum_{p=1}^{M} \sum_{\substack{\pi \in \sigma_p \wedge \\ B_p^{\pi} \neq 0}} \mathcal{W}_\infty^{p,\pi} \tag{19}$$

Since the critical path is cp with total workload $\mathcal{W}_\infty^{p,\pi}$ belonging to permutation π, the actual workload is bounded as follows: $\mathcal{W}^{p,\pi}(\gamma) \leq \mathcal{W}_\infty^{p,\pi}$ and we have:

$$B_p^{\pi} \cdot \mathcal{O}_p^{\pi(p)} \leq \mathcal{W}_\infty^{p,\pi}$$

If there is at least one processor idle ($p < M$) and there are still tasks that we need to execute, then they must be restricted by their dependencies. On unrelated multiprocessors, due to the different task-processor mappings, the execution time of any path can vary. So it is not clear which path is going to determine the makespan of the DAG. To identify the worst-case scenario to find a safe upper bound on the makespan, we consider two pessimistic but safe characteristics of the DAG. First, we consider the path with the largest workload that will guide the length of the schedule. Second, with the use of heterogeneity, we consider the worst-case mapping among all the tasks that would lead to the largest unused capacity throughout the execution. The largest unused capacity throughout the execution depends on two factors: 1) the unused processors and 2) the duration that these processors are idle. More precisely, to find the duration of the critical path workload, we assume, based on Lemma (1), that a task that belongs to the critical path is executing at speed $\mathcal{O}_p^{\pi(p)}$. If $\mathcal{O}_p^{\pi(p)}$ is busy then the unused capacity is given by $S_M^{\pi} - S_p^{\pi}$. With the heterogeneity given by Eq. (12), we can find the worst-case for a specific permutation because it maximizes these two factors among all the task-processor mappings. By replacing the $\mathcal{O}_p^{\pi(p)}$ we have:

$$\implies B_p^{\pi} \cdot \frac{S_M^{\pi} - S_p^{\pi}}{\lambda^{\pi}} \leq \mathcal{W}_\infty^{p,\pi} \tag{20}$$

By Eq. (19) and since for an arbitrary π it holds that $\lambda^{\pi} \leq \lambda$ we have:

$$\sum_{p=1}^{M} \sum_{\pi \in \sigma_p} B_p^\pi \cdot \frac{S_M^\pi - S_p^\pi}{\lambda} \leq \sum_{p=1}^{M} \sum_{\substack{\pi \in \sigma_p \wedge \\ B_p^\pi \neq 0}} \mathcal{W}_\infty^{p,\pi}$$

$$\sum_{p=1}^{M-1} \sum_{\pi \in \sigma_p} B_p^\pi \cdot \frac{S_M^\pi - S_p^\pi}{\lambda} \leq \mathcal{W}_\infty$$

Equivalently,

$$\sum_{p=1}^{M-1} \sum_{\pi \in \sigma_p} B_p^\pi \cdot (S_M^\pi - S_p^\pi) \leq \lambda \cdot \mathcal{W}_\infty \tag{21}$$

During B_p^π the p processors are busy with platform capacity S_p^π, which means that after B_p^π time units, the amount of workload that is done is $B_p^\pi \cdot S_p^\pi$. Since the application completes when no processor is busy, the total workload is given by $\mathcal{W}_1 = \sum_{p=1}^{M} \sum_{\pi \in \sigma_p} B_p^\pi \cdot S_p^\pi$. By adding this term in both sides in Eq. (21), we get:

$$\sum_{p=1}^{M-1} \sum_{\pi \in \sigma_p} [B_p^\pi \cdot (S_M^\pi - S_p^\pi) + B_p^\pi \cdot S_p^\pi] + \sum_{\pi \in \sigma_p} B_p^\pi \cdot S_M^\pi \leq \mathcal{W}_1 + \lambda \cdot \mathcal{W}_\infty$$

$$\sum_{p=1}^{M-1} \sum_{\pi \in \sigma_p} B_p^\pi \cdot S_M^\pi + \sum_{\pi \in \sigma_p} B_p^\pi \cdot S_M^\pi \leq \mathcal{W}_1 + \lambda \cdot \mathcal{W}_\infty$$

$$\sum_{p=1}^{M} \sum_{\pi \in \sigma_p} B_p^\pi \cdot S_M^\pi \leq \mathcal{W}_1 + \lambda \cdot \mathcal{W}_\infty$$

Based on the definition of S_M, given by Eq. (10) we have $S_M^\pi \geq S_M$ and by the definition of B_p given by Eq. (18), we have:

$$\implies \sum_{p=1}^{M} B_p \cdot S_M \leq \mathcal{W}_1 + \lambda \cdot \mathcal{W}_\infty$$

$$\sum_{p=1}^{M} B_p \leq \frac{\mathcal{W}_1 + \lambda \cdot \mathcal{W}_\infty}{S_M}$$

Since the scheduler is work-conserving it holds that $T_M^{\text{Comb}} = \sum_{p=1}^{M} B_p$ and consequently:

$$T_M^{\text{Comb}} \leq \frac{\mathcal{W}_1 + \lambda \cdot \mathcal{W}_\infty}{S_M}$$

This completes the proof of Lemma (2).

For the example that is given in Figure 2 we replace the parameters to the equation given in Lemma (2) and we get the makespan $T_M^{\texttt{Comb}} = 7.28$.

5.3 Efficient makespan

The calculation of the capacity and heterogeneity of the platform using the permutation-based parameters has exponential time complexity. With Lemmas (3) and (4), we prove that the permutation-independent parameters always provide more pessimistic minimum capacity and heterogeneity. The proofs are given in Appendix A. The final formula of the efficient makespan uses the total workload and the workload of the longest path presented in Section 2 and the minimum capacity and heterogeneity that are independent of the permutations that were presented in Sections 4.2 and 4.3, respectively.

Lemma 3 *The $S_M^{'}$ is always less or equal to the minimum capacity S_M between the different permutations.*

$$S_M \geq S_M^{'} \tag{22}$$

Lemma 4 *The $\lambda^{'}$ is always greater or equal to the maximum heterogeneity λ between the different permutations.*

$$\lambda \leq \lambda^{'} \tag{23}$$

Theorem 1 *Efficient makespan (`Fast`): The makespan of a DAG executed on an unrelated multiprocessor platform is given by.*

$$T_M^{\texttt{Fast}} \leq \frac{\mathcal{W}_1 + \lambda^{'} \cdot \mathcal{W}_\infty}{S_M^{'}} \tag{24}$$

Proof From Lemma (3) and (4) it follows that the makespan calculated by using $\lambda^{'}$ and $S_M^{'}$ is always larger in comparison to the makespan calculated from Lemma (2). Since the makespan $T_M^{\texttt{Comb}}$ from Lemma (2) is safe, the makespan $T_M^{\texttt{Fast}}$ given by Eq. (24) is also safe (i.e., an upper bound).

We compare the upper bound on the makespan that we computed to the optimal schedule length, i.e., minimum completion time, denoted by `OPT` to theoretically evaluate how much pessimism is introduced to our makespan computation compared to `OPT`. Finding the `OPT` is intractable (Garey and Johnson, 2002), and we compute a lower bound on the `OPT` as follows. First, we optimistically assume that the $\mathcal{GHE}$ finds a processor that provides speed one for every task of the DAG. Thus, each task is executed for its minimum WCET. Let an upgraded DAG, denoted by $\hat{G}$, be an isomorphic DAG to G, meaning that $V = \hat{V}$ and $E = \hat{E}$, where for every task instead of having M WCETs, each task has only one WCET, the c_{min}^i. Because its task has one WCET, the homogeneous setup Graham (1969); Brent (1974); Blumofe and Leiserson (1999) can be applied. A lower bound on the optimal schedule length is computed by $LB = \max\{\hat{\mathcal{W}}_\infty, \frac{\hat{\mathcal{W}}_1}{M}\}$ for the upgraded DAG $\hat{G}$, which is also a lower bound for the original DAG G.

Corollary 1 *The makespan given by Th. 1 is* $(\frac{M+\lambda^{'}}{S^{'}_{M}})$ *times larger than the optimal schedule length (*`OPT`*):*

Proof From Eq. (24), given by Theorem (1), we have:

$$T^{\texttt{Fast}}_{M} \leq \frac{\mathcal{W}_1 + \lambda^{'} \cdot \mathcal{W}_\infty}{S^{'}_{M}}$$

$$T^{\texttt{Fast}}_{M} \leq \frac{\mathcal{W}_1}{S^{'}_{M}} + \frac{\lambda^{'} \cdot \mathcal{W}_\infty}{S^{'}_{M}}$$

$$T^{\texttt{Fast}}_{M} \leq \frac{M}{S^{'}_{M}} \cdot \frac{\mathcal{W}_1}{M} + \frac{\lambda^{'}}{S^{'}_{M}} \cdot \mathcal{W}_\infty$$

By definition $LB \leq \texttt{OPT}$, so it holds that $\mathcal{W}_\infty \leq \texttt{OPT}$ and $\frac{\mathcal{W}_1}{M} \leq \texttt{OPT}$.

$$\implies T^{\texttt{Fast}}_{M} \leq \frac{M}{S^{'}_{M}} \cdot \texttt{OPT} + \frac{\lambda^{'}}{S^{'}_{M}} \cdot \texttt{OPT}$$

$$T^{\texttt{Fast}}_{M} \leq (\frac{M+\lambda^{'}}{S^{'}_{M}}) \cdot \texttt{OPT} \tag{25}$$

The value of $(\frac{M+\lambda^{'}}{S^{'}_{M}})$ depends on the WCETs of the tasks for the processors of the platform. If all the speeds for all the tasks are one, the platform specialize to homogeneous multiprocessors (Eq. (7)) (Graham, 1969; Brent, 1974; Blumofe and Leiserson, 1999) and the value (a.k.a approximation, speed-up, and resource augmentation factor) is $(2 - \frac{1}{M})$. In conclusion, the upper bound on the makespan computed by Eq. (24) is $(\frac{M+\lambda^{'}}{S^{'}_{M}})$ times larger compared to the optimal schedule length of any scheduling heuristic that has the work-conserving property and the greediness property.

5.4 Summary

This section presents two approaches to calculate the makespan of a DAG on unrelated heterogeneous multiprocessors. Initially, we present an exhaustive permutation-based approach (`Comb`) that safely calculates the makespan and has exponential time complexity (Lemma (2)). Then we use `Comb` as the stepping stone to develop `Fast` that is given in Theorem (1) to find the makespan in polynomial time. The main advantage of the approach is its generality. The assumptions regarding the platform model can cover a wide range of heterogeneous multiprocessors. The DAG model can be applied to a broad range of parallel applications, such as OpenMP task-based parallel applications. Finally, keeping the scheduler general, assuming only that it is work-conserving also allows us to cover a broad class of schedulers and not just one scheduling policy.

6 Complexity analysis

Initially, we analyze the parameters that are common for all the proposed bounds. Next, we show that the `Comb` has exponential time complexity and `Fast` has polynomial time complexity.

`Common`: For a specific task, the minimum WCET between the processors can be calculated in $O(M)$. The critical path can be calculated in $O(|V|+|E|)$ with the use of topological sort (West et al., 2001). As a result, $\mathcal{W}_\infty$ can be calculated with $O(\texttt{max}\{(|V|+|E|),(M))\}$ time complexity. The total work is the sum of the minimum WCET of all tasks, so $\mathcal{W}_1$ is computed with $O(\texttt{max}\{N,M\})$ time complexity.

`Comb` : The parameter σ_p of λ and the parameter S_M are calculated by enumerating all the permutations of the tasks (N) to processors (M). Thus, σ_p is of size $O(N^M)$ and as a result `Comb`, given by Lemma (2), has a time complexity of $O(N^M)$. So, the exhaustive approach has exponential time complexity.

`Fast`: The makespan $T_M^{\texttt{Fast}}$ is given by Theorem (1) and it uses the parameters $S_M^{'}$ and $\lambda^{'}$ that are independent of the permutations and can be calculated in polynomial time. Initially, $\mathcal{O}^i$ requires time $O(M \cdot log(M))$ to sort the array of speeds. We have to calculate $\mathcal{O}^i$ for all the tasks, so the time complexity is $O(N \cdot M \cdot log(M))$ using an efficient sorting algorithm like Heapsort. However, accessing $\mathcal{O}_x^i$ and $\overline{\mathcal{O}}_x^i$ requires constant time because the array is sorted. Next, the complexity of computing $S_M^{'}$ is $O(\texttt{max}\{N,M\})$ because to identify the minimum requires $O(N)$ and the sum includes all the processors (M). Furthermore, the calculation of the heterogeneity requires the parameter $idle_x^i$ that has time complexity $O(\texttt{max}\{N \cdot M\})$, because the maximum requires $O(N)$ and the sum is over at most $M-1$ iterations. Finally, the heterogeneity $\lambda^{'}$ uses $idle_x^i$ together with two maximum operations, that can be calculated in $O(N)$ and $O(M)$, respectively. So, $\lambda^{'}$ is calculated with time complexity $O(\texttt{max}\{N^2,M^2\})$, which is polynomial. As a result, for the `Fast`, we have $O(\texttt{max}\{(|V|+|E|),(M)\}+\texttt{max}\{N,M\}+\texttt{max}\{N,M\}+N \cdot M \cdot log(M)+\texttt{max}\{N^2,M^2\})$, that is, $O(\texttt{max}\{N^2,M^2\})$, which is polynomial in time complexity.

7 Evaluation

To quantitatively evaluate the proposed makespan calculation, Section 7.1 presents the simulation framework, and Section 7.2 presents the simulation results for different parameters of our model for four OpenMP parallel applications and synthetic DAGs.

7.1 Simulation framework

First, we present the method by which we model the DAGs of the applications and the synthetic workloads. Next, we describe the simulator that is

used to calculate the makespan. Finally, we describe the configuration of the applications and the evaluation metrics used.

7.1.1 DAG modeling

The WCET of a task is generated by adding a randomly generated value to the c^i_{min} (minimum WCET between the different processor types). With the parameter $Limit$, we limit the range of the randomly generated values. More formally, the WCET of every task is given by, $c^i_t = c^i_{min} + Rand(0, Limit)$. The exact value of c^i_{min} is stated in the experimental section. We perform two types of experiments where we consider real applications and synthetic DAGs.

Applications: We model the DAG of four parallel, task-based OpenMP applications from the BOTS benchmark suite (Duran et al., 2002): Fibonacci, Sort, Strassen, and FFT. Fibonacci has a tree-like structure and is a good representative of many recursive applications. It is simple and is very helpful for the understanding of the parallel execution of the tasks. Sort is a common operation in almost all fields of computing. Strassen is an efficient matrix-multiplication algorithm that is used in many scientific applications. Finally, FFT is used in signal and image processing. The analysis of the OpenMP code for each application is performed manually. Then the applications are implemented in our simulation framework to generate the DAG automatically. Initially, we categorize the parts of the code based on their functionality, and we introduce three nodes:

- **Spawn nodes**: The keyword *#omp pragma task* of a loop generates multiple tasks. The spawn node models the cost of parallel work generation.
- **Basic nodes**: It models the execution time of a sequentially executed code, which is the actual work of the parallel application.
- **Synchronization nodes**: We use synchronization nodes to model the *#omp pragma taskwait*. A Synchronization node models the cost (in time) of the synchronization.

For Fibonacci, Strassen, and FFT, the structure of the DAG depends on the input size. The structure of the DAG for Fibonacci depends on the actual value, and for Strassen and FFT, it depends on the array size. For Sort, the DAG structure is data-dependent; for the same array size but different actual data, we can have different DAGs. Previous work introduced conditional nodes to express the alternative execution paths. We use the method in (Baruah et al., 2015b) to transform the conditional DAG to a non-conditional worst-case DAG for Sort. An example of DAG modeling can be found in (Voudouris et al., 2017).

The applications under analysis have thousands of tasks. However, we note that the applications have only a few different tasks that perform the same function, and, as a result, they have the same set of WCETs. Tasks with the same WCET for the various processors will lead to the same permutations. Consequently, we need to calculate all the permutations only based on the

unique tasks, which, in practice, has exponential time complexity to the number of unique tasks rather than the number of tasks. For example, Fibonacci has 32836 tasks for input 20, but there is only one unique task that calculates the Fibonacci numbers. Similarly, for Sort, FFT, Strassen, there are 2, 3, and 1 unique task, respectively. In (Chronaki et al., 2015), they consider OmpSs applications, which are similar to the OpenMP applications. These applications have few unique tasks compared to the total number of tasks: Cholesky factorization, QR factorization, Heat diffusion, and Integral Histogram have 4, 4, 3, and 2 unique tasks, respectively, for DAGs with a few thousands of tasks. Although there are three categories of each node (spawn, base, and synchronization), the total number of possible pairs of tasks and node categories is significantly fewer than the total number of tasks.

Synthetic DAGs: A synthetic DAG is modeled by following a similar structure of the applications. We generate a fully-balanced tree together with the mirror tree for the *Sync* nodes. The maximum degree of the *Spawn* nodes, and the maximum height of the DAG can be set as parameters. A time budget is assigned to every *Spawn* node, which is responsible for distributing it to its child nodes and the corresponding *Sync* node to get the desirable $\mathcal{W}_1$ and $\mathcal{W}_\infty$ characteristics of the DAG. Next, the number of task types is given as a parameter to the DAG. We randomly generate WCETs with the use of the *Limit* parameter with the same approach that we use for the real applications.

7.1.2 Simulator

The simulator is event-based, where an event is considered the completion of the execution of the tasks, and we implement the scheduler described in Section 2. The real applications have many tasks, so to avoid state-space explosion, we generate the DAG gradually. We follow the schedule of the DAG, assuming that all tasks are executed for their WCET, and we monitor its execution for two independent schedules/runs. First, we schedule the DAG under consideration with infinite processors (in practice: INT_MAX, in C++) to calculate the $\mathcal{W}_1$ and $\mathcal{W}_\infty$ parameters given in Section 2. Next, for the second schedule/run, we set the number of processor types and the total number of processors that we want to test. We make sure that there is at least one processor of each type, and we use random assignment of the processors to the processor types. We schedule the DAG, and we monitor the unique tasks to determine the capacity and the heterogeneity for the `Fast` given and for `Comb` makespan given in Sections 4.2 and 4.3. Based on Lemma (2) and Theorem (1), we use the parameters provided by the simulator to calculate the makespan for $T_M^{\texttt{Comb}}$ and $T_M^{\texttt{Fast}}$. The schedule-length of the second run (Sim) is an instance of the DAG execution and cannot be used as a safe estimation of the makespan due to timing anomalies; however, it can be seen as a lower bound on the best achievable makespan.

7.1.3 Configuration and evaluation metrics

Table 3 presents the configuration of the applications. Initially, the c^i_{min} of the *Spawn*, *Base* and *Sync* are set to 300, 400 and 100 time units. The columns are the applications (Fibonacci, Sort, Strassen, and FFT). The first row is the input of the applications and the second row is the total number of nodes that the applications have. The third row is the total work ($\mathcal{W}_1$), and the fourth row is the workload of the critical path ($\mathcal{W}_\infty$) of the applications. The fifth row shows the ratio of the workload of the critical path to the total workload, and the last row shows the number of unique tasks.

	Fib	Sort	Strassen	FFT
Input	20	32768	512	8192
#Nodes	32836	16043	22410	23748
$\mathcal{W}_1$	8756400	4403300	7843300	6221400
$\mathcal{W}_\infty$	8000	14900	2500	51020
$\frac{\mathcal{W}_\infty}{\mathcal{W}_1}$	0.0009	0.003	0.0003	0.008
#Unique tasks	3	6	3	9

Table 3: Application configurations

To the best of our knowledge, no other related work provides a closed-form solution for the makespan calculation and an exact makespan of parallel applications modeled as DAGs on an unrelated multiprocessor platform. For our simulations, we use the following evaluation metrics:

Tightness: The tightness is defined as the ratio $T^{\texttt{Comb}}_M/T^{\texttt{Fast}}_M$. The exhaustive makespan calculations $T^{\texttt{Comb}}_M$ given by Lemma (2) is compared to the $T^{\texttt{Fast}}_M$ makespan given by Theorem (1).

Pessimism: We derive a lower bound on the makespan by simulating the parallel applications' actual execution with the $\mathcal{GHE}$ scheduler, where all the tasks are executed for their WCET. Let Sim be the schedule length of the execution. The pessimism of our approach is defined as the ratio of $Sim/T^{\texttt{Fast}}_M$. Note that even the optimal way to find the makespan has a length not smaller than Sim.

All the experiments are performed 100 times, and we report the average.

7.2 Quantitative results

$T^{\texttt{Fast}}_M$ is proven to be a safe makespan by showing that it is always greater than $T^{\texttt{Comb}}_M$. As a result, our evaluation needs to quantify the overestimation introduced to avoid the exponential time complexity of the $T^{\texttt{Comb}}_M$ approach. Consequently, the closer the estimation of $T^{\texttt{Fast}}_M$ is to the estimation of $T^{\texttt{Comb}}_M$, the better is the estimation. Next, by comparing our proposed approach with that of the simulation of the execution, we try to quantify the pessimism that is introduced compared to the best achievable makespan estimation. Ideally,

$T_M^{\texttt{Fast}}$ and $T_M^{\texttt{Comb}}$ are equal and as close as possible to the lower bound of the best achievable makespan. By using the evaluation metrics defined in Section 7.1.3, we present the simulation results concerning different parameters of our model. Sections 7.2.1–7.2.3 show the results considering the DAG of the applications. Sections 7.2.4 and 7.2.5 present the result of synthetic DAGs.

7.2.1 Impact of changing the number of processors

Figure 7a presents the tightness (Y-axis on the left-hand side) and the pessimism (Y-axis on the right-hand side) of Fibonacci, Sort, Strassen, and FFT as a function of the number of processors (M), where the number of processor types is up to $min\{8, M\}$. The points without a dashed line correspond to the tightness of the makespan. The points with a dashed line correspond to the pessimism. In this graph, the closer to one the values are, the better is the tightness, and the less is the pessimism.

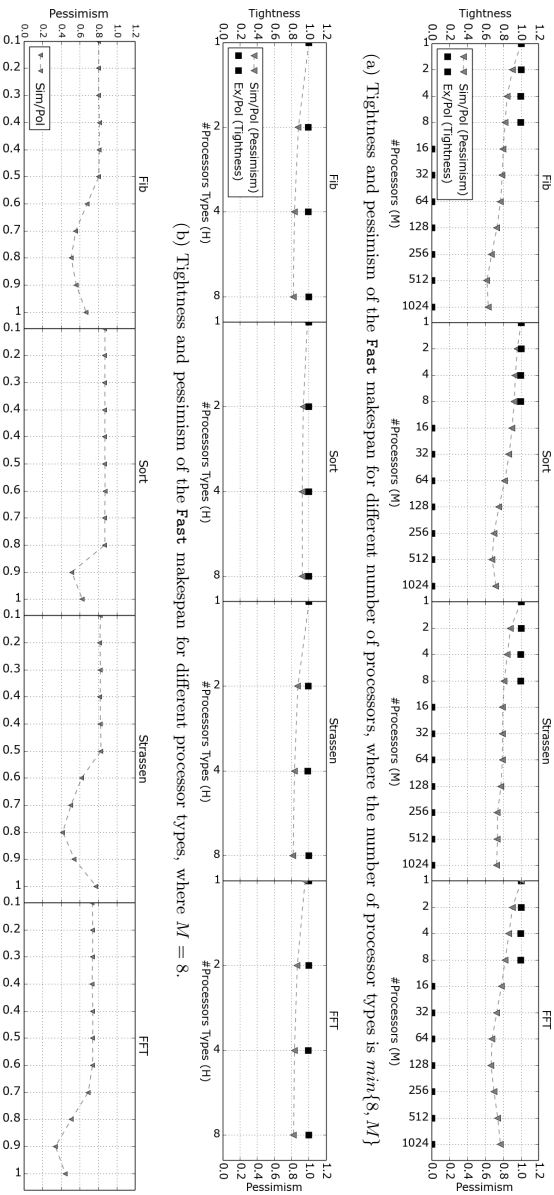
The makespan calculation of $T_M^{\texttt{Fast}}$ has polynomial time complexity, so we generate the results for up to a total of 1024 processors. On the contrary, the makespan calculation of $T_M^{\texttt{Comb}}$ is the permutation-based approach which has exponential time complexity. With our simulation setup, we can simulate only up to 8 processors. The *Limit* is set to 100 for this experiment.

Initially, it can be seen that for one processor, all the approaches are equal to $\mathcal{W}_1$. Furthermore, we can see that for up to eight processors, the tightness (the overestimation of the makespan) of $T_M^{\texttt{Fast}}$ compared to $T_M^{\texttt{Comb}}$ is less than 1% on average and up to 1.2% greater for all the applications. We have performed the same simulations, but with *Limit* equal to 500 and 1000 (not shown in the plots). The average tightness of the makespan is slightly higher than 1% and up to 3%. Next, it can be noted that by increasing the number of processors exponentially, the pessimism increases linearly. Compared to Sim, the pessimism we have averaged from 25% up to 62%.

7.2.2 Impact of changing the number of processor types

Figure 7b presents the tightness and the pessimism for the different number of processor types for eight processors and the four applications. The horizontal axis is the number of processor types for the four applications, the left vertical axis is the tightness, and the right vertical axis is the pessimism. The *Limit* for the random generation of the WCET is set to 100.

The tightness of $T_M^{\texttt{Fast}}$, compared to the two permutation-based approaches, is, on average, 1% and maximally 1.3%. Consequently, the margin between the polynomial and the exponential approach is not significant. Since we do not distinguish between the processor types for the calculation of heterogeneity and capacity in the polynomial approach but we consider the total number of processors, and it is expected to have similar behavior with the results shown in Figure 7a. Next, we note that by increasing the number of processor types while the total number of processors remains the same, the pessimism increases



(a) Tightness and pessimism of the `Fast` makespan for different number of processors, where the number of processor types is $min\{8, M\}$

(b) Tightness and pessimism of the `Fast` makespan for different processor types, where $M = 8$.

(c) Impact of making tasks incompatible to a processor type that have speed smaller than a threshold ($M = 32$) select from 4 different processor types. Each task has four different WCETs.

Fig. 7: Simulation results of the applications

since a relatively smaller number of tasks are now executing with a speed of one, which leads to a longer makespan.

Compared to *Sim*, we have, on average, 13% and up to 23% more pessimism. *Sim* is a lower bound on the optimal makespan, so if an exact makespan can be calculated for parallel applications, which is very unlikely to happen, our analysis can still provide an upper bound on makespan, which is at most 23% longer than the optimal makespan. Therefore, we think our approach to finding the makespan using T_M^{Fast} is quite effective for applications that we have considered from the BOTS benchmark suite.

7.2.3 Impact of task-processor compatibility

Whether a task is compatible with a processor type or not is determined using a threshold speed for each experiment. If the initial speed of a task on a given processor is smaller than the threshold, its speed on that processor is set to zero ($\delta_t^i = 0$).

Figure 7c shows the compatibility of the tasks to the processors. The horizontal axis presents the speed threshold for Fibonacci, Sort, Strassen, and FFT. The vertical axis shows the pessimism of T_M^{Fast} with respect to *Sim*. The platform has 32 processors and four processor types. *Limit* is set to 100 for this experiment.

Initially, for threshold 0.1, all the applications have pessimism only around $1/0.8 = 25\%$, i.e., the computed makespan is no more than 1.25 times greater than the optimal. In such a case, the scheduler can almost always find some compatible idle processor due to a relatively low threshold speed. Next, it can be seen that for all the applications, as the threshold increases, i.e., relatively more incompatible tasks, the pessimism initially remains constant and then increases since fewer compatible processors are available for the tasks to execute.

Next, we note that the pessimism starts to increase for Fibonacci and Strassen from speed threshold 0.5 while for Sort and FFT, the tightness begins to decrease after 0.8 and 0.7, respectively. Since Fibonacci and Strassen have fewer task types, 3 task types each, compared to Sort and FFT that have 6 and 9, respectively, more tasks are characterized as incompatible for Sort and FFT. As a result, more tasks have fewer processors to be executed for Sort and FFT. For Fibonacci and Strassen, tightness reaches its minimum value at 0.8 and for Sort and FFT at 0.9. There are many incompatible processors at that point, but because the platform has many processors, the scheduler can find available processors to schedule the tasks in parallel. However, we can see that the pessimism decreases for high thresholds since the scheduler (i.e., simulated schedule) cannot find available processors to schedule the tasks, and the total execution of the schedule increases. Consequently, the pessimism compared to *Sim* decreases.

7.2.4 Impact of processor heterogeneity

To analyze in more detail the variation of the WCET of a task among the different processor types, we consider a synthetic DAG, and we vary the *Limit* factor. Figure 8 presents the tightness and the pessimism for different values of the *Limit*. The horizontal axis is the *Limit*, the left vertical axis presents the tightness, and the right vertical axis is the pessimism. The platform has four processors and two processor types. The synthetic DAG has $\mathcal{W}_1 = 191400$ and $\mathcal{W}_\infty = 5800$ for 938 nodes and 3 task types with ratio $\frac{\mathcal{W}_\infty}{\mathcal{W}_1} = 0.03$. Note that this ratio is one to two orders of magnitude higher than the BOTS applications, so the impact of heterogeneity should be higher. Note that with $Limit = 1000$, we can have a variation on the WCET from $2.5x$ to $10x$ for *Spawn* and *Sync* nodes, respectively, that have 400 and 100 time units for their e_i^{min} values. We intentionally use extreme values to expose the limitations of T_M^{Fast}.

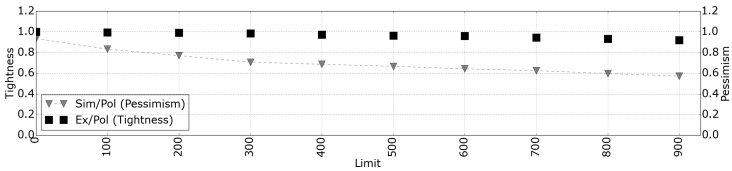


Fig. 8: Tightness and pessimism for different variations of the WCET.

By increasing *Limit*, which can be seen as making the platform more heterogeneous, the tightness of the T_M^{Fast} approach decreases. On average, we have 5% and a maximum 11% less tight makespan compared to exhaustive approaches. Such an increase in the makespan is due to the calculation of the heterogeneity λ', which is calculated between all the tasks. T_M^{Comb} uses λ which is calculated based on all of the permutations of tasks. We can see that the pessimism of T_M^{Fast} compared to *Sim* increases as *Limit* increases since more processors would have a lower speed. As a result, the makespan of T_M^{Fast} increases. Compared to *Sim*, we have on average 51% and up to 74% more pessimism. Note that although such values may be quite high for our analysis, we would like to stress that the degree of heterogeneity for higher *Limit* is quite pessimistic for many practical heterogeneous platforms.

7.2.5 Impact of application characteristic

For this experiment, we characterize a DAG by $\mathcal{W}_1$ and $\mathcal{W}_\infty$ only, and we vary the characteristic (i.e., $\frac{\mathcal{W}_\infty}{\mathcal{W}_1}$) of an application. Note that $\frac{\mathcal{W}_\infty}{\mathcal{W}_1}$ is within $(0, 1)$. If $\frac{\mathcal{W}_\infty}{\mathcal{W}_1} \approx 0$, then it means that the length of the critical path is much smaller in comparison to that of the total work (more dense graph). If $\frac{\mathcal{W}_\infty}{\mathcal{W}_1} \approx 1$, then it means that the length of the critical path is very close to the total work (more

sparse graph). Figure 9 shows the tightness for the proposed methods where we keep the value of $\mathcal{W}_1$ constant and vary the value of $\mathcal{W}_\infty$. The horizontal axis shows different $\frac{\mathcal{W}_\infty}{\mathcal{W}_1}$ ratios (significantly larger compared to the applications), and the vertical axis shows the tightness.

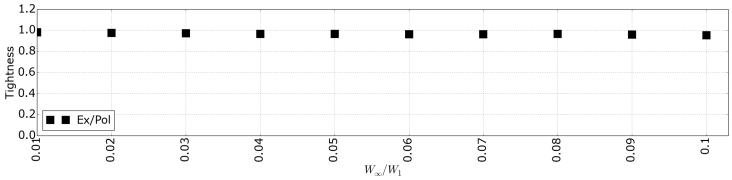


Fig. 9: Comparison of the proposed methods for different characteristic of the synthetic DAG by varying the $\frac{\mathcal{W}_\infty}{\mathcal{W}_1}$ ratio.

Initially, for both cases, the makespan increases since the critical path increases. Next, the tightness decreases as the $\frac{\mathcal{W}_\infty}{\mathcal{W}_1}$ increases since by increasing the $\mathcal{W}_\infty$, the impact of the heterogeneity increases. $T_M^{\texttt{Fast}}$ has, on average, 6% and at maximum 16%, less tight makespan compared to the exhaustive approach.

7.3 Summary

From the simulation results, we can see that $T_M^{\texttt{Fast}}$ provides tight makespan estimation compared to $T_M^{\texttt{Comb}}$ and with low pessimism compared to the simulation of the execution Sim. We quantitatively verify the intuition by increasing the number of incompatible processors the makespan increases, which shows that the parallelism is restricted and leads to a larger estimation of the makespan. Next, we have seen that increasing the variation of the WCET across the different processor types leads to higher pessimism. Finally, by increasing the critical path's workload and total workload ratio, the makespan increases because less parallelism is available.

8 Comparison with similar approaches

This section compares our model with models in the literature that make more specific assumptions regarding multiprocessor platforms and applications. Initially, we compare our approach to approaches that assume the homogeneous and related multiprocessor model. Next, we compare our approach to a more specific application and platform model where Typed DAGs (Han et al., 2019) is assumed.

8.1 Homogeneous and related multiprocessor models

Table 4 shows the specializations of the proposed formula to formulas proposed and used in related work. If $\delta_t^i = 1$, for any task of the application and any processor, the multiprocessor platform is homogeneous. For the platform capacity and heterogeneity it holds that $S_M^{'} = M$ and $\lambda^{'} = (M-1)$, respectively. As a result, the proposed formula becomes the same formula ($\frac{\mathcal{W}_1 + \frac{(M-1)}{1} \cdot \mathcal{W}_\infty}{M}$) developed in (Graham, 1969; Brent, 1974) and used extensively in previous works, for example (Blumofe and Leiserson, 1999; Melani et al., 2015). Similarly, by assuming the same speeds for the processors for all the tasks, the formula is the same as the formula proposed in the context of the related multiprocessor model by (Jiang et al., 2017).

	U **This work**	H Graham (1969)	R Jiang et al. (2017)
Heterogeneity	$\lambda^{'}$	$M-1$	$\lambda^R = \lambda^{'}$
Capacity	$S_M^{'}$	M	$S_M^R = S_M^{'}$
Makespan	$\frac{\mathcal{W}_1 + \lambda^{'} \cdot \mathcal{W}_\infty}{S_M^{'}}$	$\frac{\mathcal{W}_1 + \frac{(M-1)}{1} \cdot \mathcal{W}_\infty}{M}$	$\frac{\mathcal{W}_1 + \lambda^R \cdot \mathcal{W}_\infty}{S_M^R}$

Table 4: Specializations of the (U)nrelate multiprocessor model to (H)omegeneous, (R)elated multiprocessor models.

8.2 Typed DAG application model

In the work of (Han et al., 2019), typed DAGs (i.e., every task is compatible with one processor type) are assumed, and two bounds are proposed to estimate the makespan. The proposed bounds strictly dominate the used baseline in (Jaffe, 1980) and, through simulation, outperforms the work by (Yang et al., 2016) significantly. The first approach (NEW-B-1) is a generalization of the (Graham, 1969) for typed DAGs. The second bound (NEW-B-2) explores the structure of the DAG and provides a tighter makespan.

By restricting the compatibility of the tasks, the parallelism is reduced because fewer processors are available to execute every task. Also, the critical path may be spread to different processor types. As a result, all the nodes that belong to processor type t_1 can interfere with tasks that belong to the critical path and are executed on processor type t_2. The proposed analysis (Jaffe, 1980; Han et al., 2019) reflects this problem by assuming no parallelism between tasks executed on different processor types. It sums, i.e., serializes, the execution of the nodes that belong to different types. This assumption limits parallelism significantly. For example, for any number of processors that a multiprocessor has, if there is only one processor of each type, all the approaches are equal to the sequential execution.

We can model Typed DAGs with our settings by assuming for a task τ^i that $\delta_t^i = 1$ for the compatible processors and $\delta_t^i = 0$ for the non-compatible

	$T_M^{\texttt{Fast}}$ **This work**	$T_M^{\max(\pi)}$ **This work**	NEW-B-1 Han et al. (2019)
Makespan	$\frac{\mathcal{W}_1+(M-1)\cdot\mathcal{W}_\infty}{M_{\min}}$	$\frac{\mathcal{W}_1+(M_{\min}-1)\cdot\mathcal{W}_\infty}{M_{\min}}$	$\sum_{t\in S}\frac{\mathcal{W}_1^t+(M_t-1)\cdot\mathcal{W}_\infty^t}{M_t}$

Table 5: Specializations of the unrelated to the typed-DAG application model

processors. Let M_t denote the number of processors of type t and let $M_{\min}$ be the minimum number of processors between the different processor types. Table 5 presents the calculations for the typed DAG model. By trivially applying the $T_M^{\texttt{Fast}}$ approach, for any typed DAG the platform capacity is $S_M^{'} = M_{\min}$ and heterogeneity is $\lambda^{'} = M-1$. So, the makespan of $T_M^{\texttt{Fast}}$ is more pessimistic than (Jaffe, 1980) and, as a result, also than the (NEW-B-1) and (NEW-B-2) from (Han et al., 2019). To reduce the pessimism, we can find the makespan by computing the capacity and the heterogeneity from the same permutation. More precisely, first, we compute the makespan for all the possible permutations. Then we find the maximum makespan among all the permutations, denoted as $T_M^{\max(\pi)}$, and the platform capacity is $M_{\min}$ and heterogeneity is $M_{\min}-1$. This approach has exponential time complexity since all the permutations need to be searched. The $T_M^{\max(\pi)}$ is better than $T_M^{\texttt{Fast}}$ but still more pessimistic than (NEW-B-1) and (NEW-B-2). By assuming that the mapping of the tasks is known, we can calculate $\mathcal{W}_\infty^t$ and $\mathcal{W}_1^t$ for each M_t. By serializing the execution between the types and applying our formula, we find the same formula as for (NEW-B-1), which is more pessimistic than (NEW-B-2).

9 Related work

In (Graham, 1969; Brent, 1974), a makespan calculation is presented for parallel applications modeled as DAGs executed on homogeneous multiprocessors. The work in (Blumofe and Leiserson, 1999) extends this bound in the Cilk programming model context. The Cilk-based parallel applications are modeled with a restricted version of DAGs, and the bound is extended to cover the work-stealing scheduler. In (Voudouris et al., 2017; Chen et al., 2019) a formally proven timing anomaly-free dynamic scheduler is introduced that provides tighter and more scalable, for the number of tasks and number of processors, makespan estimations. The results of (Voudouris et al., 2017; Chen et al., 2019) cannot be trivially applied to unrelated multiprocessors because the DAG can have different schedule lengths depending on the task-processor mapping.

In (Bender and Rabin, 2000), the scheduler of Cilk (Blumofe and Leiserson, 1999) is adapted for related heterogeneous systems. They provide a makespan calculation methodology for Cilk-based applications that can be modeled as DAGs, and a makespan is introduced. Our approach considers the unrelated multiprocessor model, which is a more general model for the underlying platform.

The work in (Sih and Lee, 1993; Topcuoglu et al., 2002) considers *static* scheduling of applications modeled as DAGs on unrelated multiprocessor plat-

forms, and the goal is to minimize the schedule length. An extensive comparison of different heuristics for static scheduling on heterogeneous systems can be found in Braun et al. (2001). In contrast, our approach considers *dynamic* scheduling that can utilize the platform more efficiently to achieve load balance among the processors.

In (Lawler and Labetoulle, 1978), global scheduling of independent tasks for unrelated multiprocessor scheduling is formulated and solved as an integer linear problem. In (Andersson et al., 2010; Raravi et al., 2013) the problem of scheduling independent tasks on two types of unrelated heterogeneous multiprocessor platforms is considered, and further extension of the works in (Andersson et al., 2010; Raravi et al., 2013) can be found in (Raravi, 2014). Next, (Andersson and Raravi, 2014) assumes implicit-deadline, independent tasks, unrelated multiprocessor platforms, and shared resources, a speed-up bound of $4 \cdot (1 + \epsilon)$, where ϵ depends on the number of shared resources and the resource requests from the tasks. The assumption of shared resources enriches the applicability of the model. However, we do not address this problem in this paper, and we leave it as future work. Furthermore, the work in (Andersson and Raravi, 2016) assumes constrained-deadline independent tasks and unrelated platforms but is limited to two processor types. The problem is formulated as an ILP, and a speed-up bound of 5 is guaranteed. Next, (Baruah et al., 2019) with the ILP approach for constrained deadlines, independent tasks, unrelated multiprocessors, and partitioned scheduling, a speed-up bound of 7.83 is achieved. Our approach considers a more general application model which can exploit the parallelism that exists in the applications. In this work, we assume a single DAG, and our goal is to find the makespan which is needed for the analysis for the recurrent execution of DAGs.

Previous work for general-purpose scheduling on unrelated multiprocessors has focused on special cases of our system model either by limiting the structure of the DAG (Kumar et al., 2009) or by limiting the execution time of the tasks and their compatibility to the processors (Page, 2019). In this work, we consider DAGs where each task can execute on any processor and can have any execution time. In addition, the proposed makespan can be applied to any priority ordering of the task that has the work-conserving and the greediness property. As a result, in this work, instead of focusing on finding a carefully optimized scheduler for special cases of the application or platform models, we opt to find a makespan computation formula that is general and has broad applicability.

The estimation of the makespan of a single DAG gives us the tool to analyze multiple DAGs with the sporadic DAG model (Baruah et al., 2015a, 2012). In (Li et al., 2014; Melani et al., 2015; Pathan et al., 2018) global scheduling is assumed and analyzed for homogeneous multiprocessors. Furthermore, federated scheduling, which can be seen as a generalization of partitioned scheduling, recently has gained attention, and many recent works focus on this topic (Li et al., 2014; Jiang et al., 2017; Bhuiyan et al., 2018; Ueter et al., 2018). This paper proposes a single DAG analysis on unrelated multiprocessors, which is the first step towards the analysis of sporadic DAGs.

10 Conclusion

We propose two approaches to calculate the upper bound on the worst-case-schedule-length (makespan) for applications modeled as DAGs and executed on unrelated multiprocessors using any work-conserving scheduler. First, with an exhaustive approach, we show that `Comb` can safely establish an upper bound of the makespan. Still, its applicability is limited to small platforms and DAGs because it has exponential time complexity. We use `Comb` to build the `Fast` makespan that trades off the precision, i.e., tightness, of `Comb` to achieve polynomial time complexity.

To quantitatively evaluate the makespan of `Fast`, we model as DAGs four OpenMP task-based parallel applications and synthetic workloads. We compare `Fast` to `Comb` to determine the tightness. Based on the simulation results, the `Fast` approach finds the makespan nearly as tight as the `Comb` approach. Furthermore, we compare `Fast` with the simulation of the assumed scheduler that is a lower bound on the best-achievable makespan and we show that its estimation has low pessimism.

The main advantage of the proposed approach is its generality because it can be applied to a broad range of platforms, applications, and schedulers. The unrelated model is very expressive and can model many available platforms today using a wide range of processor types and specialized application accelerators. The DAG model is capable of capturing the behavior of many parallel applications. The scheduler is dynamic, so it can deal with a large number of fine-grain tasks that, for example, an OpenMP parallel application can have. The scheduler also supports arbitrary compatibility of the tasks to the processors. So, we can model accelerators that are designed to perform a limited set of operations more efficiently. Furthermore, the scheduler does not assume any scheduling policy, so our analysis can be applied to many well-known work-conserving schedulers from the related work. By fixing the WCET relation of the tasks to the processors, we show that the proposed makespan specializes (derives the same closed-form solution) to well-known bounds for homogeneous multiprocessors and recently developed related multiprocessors and typed DAGs.

The main limitation of the paper is the abstraction of the platform's architectural details. We do not consider any shared resources between the task. However, in practice, many hardware components, for example, memory and interconnect, are shared. The use of shared resources significantly complicates the problem, and detailed timing analysis is needed to determine the interference of the tasks. We do not address the issue of shared resources in this paper. However, we expect the shared resource timing analysis to be orthogonal with our analysis and that it would increase the applicability of the model.

To the best of our knowledge, no related work covers the combination of assumptions: DAG application model, unrelated multiprocessor model, and work-conserving scheduling. As future work, we plan to develop the analysis of multiple DAGs that are executed on unrelated multiprocessors with the use of the sporadic DAG model (Baruah et al., 2015a, 2012).

A Proofs of Lemmas (3) and (4)

This Section contains the proofs of lemmas that are used for the proof of the efficient makespan calculation (T_M^{Fast}) given by Theorem (1).

A.1 Proof of Lemma (3)

Lemma (3) states that the platform capacity $S_M^{'}$ is always less or equal to minimum platform capacity between the different permutations S_M. The proof of Lemma (3) is given as follows:

Proof Let $\pi(x)$ denotes the x^{th} task that belongs to a permutation π. For an arbitrary task $\pi(x)$ that it is executing on its x^{th} fastest processor it holds that:

$$\mathcal{O}_x^{\pi(x)} \geq \min_{i=1}^{N}\{\mathcal{O}_x^i\}$$

Since the size of the $\mathcal{O}^i$ is equal to the number of processors M it holds that:

$$\sum_{x=1}^{M}\{\mathcal{O}_x^{\pi(x)}\} \geq \sum_{x=1}^{M}\min_{i=1}^{N}\{\mathcal{O}_x^i\}$$

Since it holds for an arbitrary permutation π, it also holds for the permutation that provides the minimum value.

$$\min_{\pi\in\sigma_M}\{\sum_{x=1}^{M}\mathcal{O}_x^{\pi(x)}\} \geq \sum_{x=1}^{M}\min_{i=1}^{N}\{\mathcal{O}_x^i\}$$

From the definitions of S_M and $S_M^{'}$ given by equations (10) and (11) respectively, the statement of the lemma holds.

A.2 Proof of Lemma (4)

Lemma (4) states that the heterogeneity $\lambda^{'}$ is always greater or equal to the maximum heterogeneity λ between the different permutations. The proof of Lemma (4) follows.

Proof Let $\overline{\mathcal{O}}_x^{\pi(x)}$ be the subset of speeds that are smaller than the x^{th} fastest processor of the x^{th} task that belongs to permutation π. For an arbitrary task $1 \leq \pi(y) \leq N$ that it is executed on its y^{th} fastest processor, it holds that:

$$\mathcal{O}_y^{\pi(y)} \leq \max_{j=1}^{N}\{\mathcal{O}_y^j\}$$

Equivalently,

$$\sum_{y \in \overline{\mathcal{O}}_x^{\pi(x)}} \{\mathcal{O}_y^{\pi(y)}\} \leq \sum_{y \in \overline{\mathcal{O}}_x^{\pi(x)}} \max_{j=1}^{N}\{\mathcal{O}_y^j\}$$

By dividing both sides with the speed of the x^{th} fastest processor $\mathcal{O}_x^{\pi(x)} \neq 0$, of task $\pi(x)$ we have:

$$\frac{\sum_{y \in \overline{\mathcal{O}}_x^{\pi(x)}} \{\mathcal{O}_y^{\pi(y)}\}}{\mathcal{O}_x^{\pi(x)}} \leq \frac{\sum_{y \in \overline{\mathcal{O}}_x^{\pi(x)}} \max_{j=1}^{N}\{\mathcal{O}_y^j\}}{\mathcal{O}_x^{\pi(x)}}$$

Since the inequality holds for any processor x it holds also for the processor that maximizes the two sides of the inequality:

$$\max_{x=1}^{M}\{\frac{\sum_{y \in \overline{\mathcal{O}}_x^{\pi(x)}} \{\mathcal{O}_y^{\pi(y)}\}}{\mathcal{O}_x^{\pi(x)}}\} \leq \max_{x=1}^{M}\{\frac{\sum_{y \in \overline{\mathcal{O}}_x^{\pi(x)}} \max_{j=1}^{N}\{\mathcal{O}_y^j\}}{\mathcal{O}_y^{\pi(y)}}\}$$

Since it holds for an arbitrary task with index $\pi(x)$ that belongs to an arbitrary permutation π it also holds for any task of the application $1 \leq j \leq N$ and we have:

$$\max_{x=1}^{M}\{\frac{\sum_{y \in \overline{\mathcal{O}}_x^{\pi(x)}} \max_{j=1}^{N}\{\mathcal{O}_y^j\}}{\mathcal{O}_x^{\pi(x)}}\} = \max_{x=1}^{M}\{\frac{\sum_{y \in \overline{\mathcal{O}}_x^{k}} \max_{j=1}^{N}\{\mathcal{O}_y^j\}}{\mathcal{O}_x^{k}}\} \tag{26}$$

Since $1 \leq k \leq N$ it also holds that:

$$\max_{x=1}^{M}\{\frac{\sum_{y \in \overline{\mathcal{O}}_x^{k}} \max_{j=1}^{N}\{\mathcal{O}_y^j\}}{\mathcal{O}_x^{k}}\} \leq \max_{i=1}^{N}\{\max_{x=1}^{M}\{\frac{\sum_{y \in \overline{\mathcal{O}}_x^{i}} \max_{j=1}^{N}\{\mathcal{O}_y^j\}}{\mathcal{O}_x^{i}}\}\} \tag{27}$$

From (26) and (27) we have:

$$\max_{x=1}^{M}\{\frac{\sum_{y \in \overline{\mathcal{O}}_x^{\pi(x)}} \max_{j=1}^{N}\{\mathcal{O}_y^j\}}{\mathcal{O}_x^{\pi(x)}}\} \leq \max_{i=1}^{N}\{\max_{x=1}^{M}\{\frac{\sum_{y \in \overline{\mathcal{O}}_x^{i}} \max_{j=1}^{N}\{\mathcal{O}_y^j\}}{\mathcal{O}_x^{i}}\}\}$$

Let π' be the permutation that provides the maximum value for the maximum accumulated capacity loss. Since it holds for any π it also holds for the permutation π', we have:

$$\max_{k=1}^{M}\{\frac{\sum_{y\in\overline{\mathcal{O}}_x^{\pi'(x)}}\{\mathcal{O}_y^{\pi'(y)}\}}{\mathcal{O}_x^{\pi'(x)}}\} \leq \max_{i=1}^{N}\{\max_{x=1}^{M}\{\frac{\sum_{y\in\overline{\mathcal{O}}_x^{i}}\max_{j=1}^{N}\{\mathcal{O}_y^{j}\}}{\mathcal{O}_x^{i}}\}\}$$

Equivalently,

$$\max_{\pi\in\sigma_M}\{\max_{k=1}^{M}\{\frac{\sum_{y\in\overline{\mathcal{O}}_x^{\pi(x)}}\{\mathcal{O}_y^{\pi(y)}\}}{\mathcal{O}_x^{\pi(x)}}\}\} \leq \max_{i=1}^{N}\{\max_{x=1}^{M}\{\frac{\sum_{y\in\overline{\mathcal{O}}_x^{i}}\max_{j=1}^{N}\{\mathcal{O}_y^{j}\}}{\mathcal{O}_x^{i}}\}\}$$

From the definitions of λ, $\lambda^{'}$ and $idle_x^i$ given by the equations (13), (15) and (14), we have: $\lambda \leq \lambda^{'}$. As a result $\lambda^{'}$ is always greater than or equal to λ.

References

Andersson B, Raravi G (2014) Real-time scheduling with resource sharing on heterogeneous multiprocessors. Real-Time Systems, Springer

Andersson B, Raravi G (2016) Scheduling constrained-deadline parallel tasks on two-type heterogeneous multiprocessors. In: Proceedings of the 24th International Conference on Real-Time Networks and Systems, ACM

Andersson B, Raravi G, Bletsas K (2010) Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors. In: IEEE RTSS

ARM (2011) big.little technology: The future of mobile. White paper, `https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf`

Baruah S, Bonifaci V, Marchetti-Spaccamela A, Stougie L, Wiese A (2012) A generalized parallel task model for recurrent real-time processes. In: IEEE RTSS

Baruah S, Bertogna M, Buttazzo G (2015a) Multiprocessor scheduling for real-time systems. Springer

Baruah S, Bonifaci V, Marchetti-Spaccamela A (2015b) The global edf scheduling of systems of conditional sporadic dag tasks. In: IEEE ECRTS

Baruah SK, Bonifaci V, Bruni R, Marchetti-Spaccamela A (2019) Ilp models for the allocation of recurrent workloads upon heterogeneous multiprocessors. Journal of Scheduling

Bender MA, Rabin MO (2000) Scheduling cilk multithreaded parallel programs on processors of different speeds. In: ACM SPAA

Bhuiyan A, Guo Z, Saifullah A, Guan N, Xiong H (2018) Energy-efficient real-time scheduling of dag tasks. ACM TECS

Blumofe RD, Leiserson CE (1999) Scheduling multithreaded computations by work stealing. JACM

Braun TD, et al. (2001) A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. Journal of Parallel and Distributed computing

Brent RP (1974) The parallel evaluation of general arithmetic expressions. JACM

Chen P, Liu W, Jiang X, He Q, Guan N (2019) Timing-anomaly free dynamic scheduling of conditional dag tasks on multi-core systems. ACM Transactions on Embedded Computing Systems (TECS)

Chronaki K, et al. (2015) Criticality-aware dynamic task scheduling for heterogeneous architectures. In: ACM, ICS

Chwa HS, Seo J, Lee J, Shin I (2015) Optimal real-time scheduling on two-type heterogeneous multicore platforms. In: IEEE RTSS

Duran A, et al. (2002) Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In: ICPP

Esmaeilzadeh H, Blem E, Amant RS, Sankaralingam K, Burger D (2011) Dark silicon and the end of multicore scaling. In: IEEE ISCA

Funk S, Goossens J, Baruah S (2001) On-line scheduling on uniform multiprocessors. In: IEEE RTSS

Garey MR, Johnson DS (2002) Computers and intractability. wh freeman New York

Graham RL (1969) Bounds on multiprocessing timing anomalies. SIAM journal on Applied Mathematics

Gupta A, Im S, Krishnaswamy R, Moseley B, Pruhs K (2012) Scheduling heterogeneous processors isn't as easy as you think. In: ACM-SIAM SODA

Han M, Guan N, Sun J, He Q, Deng Q, Liu W (2019) Response time bounds for typed dag parallel tasks on heterogeneous multi-cores. IEEE TPDS

Jaffe JM (1980) Bounds on the scheduling of typed task systems. SIAM Journal on Computing

Jiang X, Guan N, Long X, Yi W (2017) Semi-federated scheduling of parallel real-time tasks on multiprocessors. IEEE RTSS

Kumar VA, Marathe MV, Parthasarathy S, Srinivasan A (2009) Scheduling on unrelated machines under tree-like precedence constraints. Springer Algorithmica

Lakshmanan K, Kato S, Rajkumar R (2010) Scheduling parallel real-time tasks on multi-core processors. In: IEEE RTSS

Lawler EL, Labetoulle J (1978) On preemptive scheduling of unrelated parallel processors by linear programming. Journal of the ACM (JACM)

Li J, Chen JJ, Agrawal K, Lu C, Gill C, Saifullah A (2014) Analysis of federated and global scheduling for parallel real-time tasks. In: IEEE ECRTS

Melani A, Bertogna M, Bonifaci V, Marchetti-Spaccamela A, Buttazzo GC (2015) Response-time analysis of conditional dag tasks in multiprocessor systems. In: ECRTS

Page DR (2019) Approximation algorithms for problems in makespan minimization on unrelated parallel machines. PhD thesis, The University of Western Ontario

Pathan R, Voudouris P, Stenström P (2018) Scheduling parallel real-time recurrent tasks on multicore platforms. IEEE TPDS

Peter Greenhalgh A (2011) Big.little processing with arm cortex-a15 and cortex-a7 improving energy efficiency in high-performance mobile platforms. White paper, `http://www.cl.cam.ac.uk/~rdm34/big.LITTLE.pdf`

Raravi G (2014) Real-time scheduling on heterogeneous multiprocessors. PhD thesis, Faculty of Engineering, University of Porto

Raravi G, Andersson B, Bletsas K (2013) Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors. Springer RTS

Sih GC, Lee EA (1993) A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. IEEE TPDS

Topcuoglu H, Hariri S, Wu My (2002) Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE TPDS

Ueter N, von der Brüggen G, Chen JJ, Li J, Agrawal K (2018) Reservation-based federated scheduling for parallel real-time tasks. In: IEEE RTSS

Voudouris P, Stenström P, Pathan R (2017) Timing-anomaly free dynamic scheduling of task-based parallel applications. In: IEEE RTAS

West DB, et al. (2001) Introduction to graph theory. Prentice hall Upper Saddle River

Yang K, Yang M, Anderson JH (2016) Reducing response-time bounds for dag-based task systems on heterogeneous multicore platforms. In: ACM RTNS

Chapter 6

Paper III

Response time analysis for globally scheduled sporadic DAGs on unrelated multiprocessors

Petros Voudouris, Per Stenström, Risat Pathan

Manuscript, 2021.

Response time analysis for globally scheduled sporadic DAGs on unrelated multiprocessors

Petros Voudouris, Per Stenström, and Risat Pathan

Chalmers University of Technology, Göteborg SE-412 96, Sweden
{petrosv,pers,risat}@chalmers.se

Abstract. This paper addresses the problem of scheduling a set of real-time sporadic DAGs with constrained deadlines on unrelated heterogeneous multiprocessor platforms. We propose a scheduler, called $\mathcal{GUM}$, that can be used to dispatch the nodes of multiple DAGs on a heterogeneous platform. The $\mathcal{GUM}$ scheduler allows the migration of the nodes from one processor to another to run them faster to exploit the platform's heterogeneity effectively. Our proposed $\mathcal{GUM}$ scheduler is general because it can easily be applied to any priority-based real-time scheduler. Based on the $\mathcal{GUM}$ scheduler, we first propose an analytical way to compute the makespan of a single DAG. Then we determine the interference of each DAG on another DAG when they are scheduled together. Based on the makespan and interference, we compute the worst-case response time of each DAG to determine whether the deadline a DAG is met or not. Our empirical evaluation demonstrates the effectiveness in meeting the real-time deadlines of randomly generated set of sporadic DAGs.

Keywords: Hard real-time scheduling · Sporadic DAGs · Unrelated heterogeneous multiprocessors · Response-time analysis.

1 Introduction

Parallel applications that are executed on heterogeneous multiprocessors can provide high performance and energy efficiency for time-critical applications. For hard real-time systems, offline schedulability analysis is necessary to guarantee that an application's behavior is time predictable. In this paper, we approach the problem of computing an upper bound on the completion time (also known as the worst-case response time [3]) for each of a collection of parallel applications with strict real-time requirements where the applications execute on a heterogeneous multiprocessor platform.

A parallel application (e.g., OpenMP [11]) can be modeled as a directed acyclic graph (DAG), where the nodes represent tasks and the edges represent dependencies among the tasks. The sporadic DAG model [3] assumes a finite number of recurring DAGs. Each DAG generates a potentially infinite sequence of releases. The first release can arrive at any time instant, and any two consecutive releases are separated at least by their minimum inter-arrival time (also called the period). Each DAG has a relative deadline that needs to be met, and

it is relative to the arrival time of every release of that DAG. The sporadic model is very suitable to model many real-time control and monitoring applications.

Previous work has shown that heterogeneous multiprocessors can provide high performance and energy efficiency [1]. A heterogeneous multiprocessor platform can be modeled by the unrelated multiprocessor platform model [3]. In the unrelated model, multiprocessors have processors of different *types*. Each task of a DAG can have a different worst-case execution-time (WCET) [17] on different processor types. The design and timing analysis of a scheduler that can efficiently utilize the platform by considering the diversity of processor types is necessary for the correct execution of real-time applications on an unrelated platform.

A dynamic (also called a global) scheduler can schedule a task of a DAG to any of the processors of a platform. We consider a global and work-conserving scheduler such that a processor is never idle if there is available work to schedule. As a result, such a scheduler can balance the work among the processors to utilize them efficiently. However, the schedulability analysis of such a scheduler for a heterogeneous platform is challenging because the task-processor mapping can change from execution to execution and can lead to different schedule lengths.

The schedulability analysis of a set of sporadic DAGs can be separated into two parts. The first part is the intra-DAG analysis, where a single DAG is analyzed in isolation, and the main challenge is to estimate its worst-case schedule length, called the makespan. The second part is the inter-DAG analysis, where the main challenge is to safely upper bound the interference that each DAG suffers from other DAGs when competing for the computing resource. By combining the makespan analysis of a single DAG and the inter-DAG interference analysis, we find the worst-case completion time, called the response time of each DAG.

The main contribution of this paper is the design of a scheduler and its schedulability analysis to guarantee offline the timeliness for sporadic DAGs that execute on unrelated multiprocessors. The scheduler is *global, work-conserving, preemptive*, and, *migrative* and it dispatches or migrates a task to the fastest available processor. The scheduler is suitable for worst-case schedulability analysis because it has a property that allows us to find the worst-case task-processor mapping that may happen during run time. The proposed scheduler is a generalization for unrelated multiprocessors of the scheduler used in [13,15] that targets homogeneous multiprocessors. Based on the schedulability analysis of the scheduler, we generalize the classical window-based schedulability analysis [2,5,3,15] for unrelated multiprocessors to find the response time of each sporadic DAG. We then specialize the response-time analysis by assuming both fixed- and dynamic-priority ordering.

Previous work that uses the unrelated multiprocessor model has focused on independent parallel tasks [6]. Some work assumes DAG as an application model but focused on related multiprocessors [14]. A few work have also focused on typed DAGs [19,16,20] where each task of a DAG can execute only one processor type. To the best of our knowledge, this paper is the first to propose a method to compute the worst-case response time for globally scheduled sporadic DAGs on unrelated multiprocessors.

Contributions. The *first contribution* of the paper is the design of a scheduler for a set of sporadic DAGs that execute on an unrelated multiprocessor platform. The scheduler is based on a global strategy and is work-conserving with migration capability so that tasks of each DAG can enjoy a higher speed as soon as a more capable processor becomes idle at run time. The *second contribution* of the paper is the schedulability analysis of sporadic DAGs that execute on unrelated multiprocessors. We generalize for unrelated multiprocessors the well established analytical tools to compute the response time of sporadic DAGs executing on homogeneous multiprocessors. Finally, the *third contribution* is the empirical evaluation of the proposed scheduler. We randomly generate synthetic DAGs and check if all their deadlines are met. The simulation results show that the well-known rate monotonic and earliest deadline first priority assignment are effective for heterogeneous multiprocessors and perform better than arbitrary priority assignment.

Paper organization: Section 2 presents the system model and useful definitions used for formal timing analysis of the proposed scheduling algorithm. Section 3 presents the pseudocode of the proposed scheduler and the greediness property that we rely on to reason about the worst-case execution behavior of the parallel applications. Based on the assumed system model and the greediness property, we in Section 4 formally present the detailed schedulability analysis to find the makespan of a single DAG. Section 5 extends the timing analysis to determine the interference that one DAG has on another DAG to find the response time of each sporadic DAG. Section 6 presents the simulation framework and the empirical results based on synthetic workloads to demonstrate the effectiveness of our proposed timing analysis in ensuring real-time constraints. Section 7 presents related work before concluding the paper in Section 8.

2 System model

We model a heterogeneous platform with the unrelated multiprocessor model [3]. Every processor can be different, and a processor type characterizes it. A platform with a total of $\mathcal{P}$ processors can have one up to $\mathcal{P}$ processor types. Without loss of generality, we index the processors from 1 to $\mathcal{P}$.

We consider a set of sporadic DAGs, denoted by Γ. The set $\Gamma = \{G^1, \dots G^{\mathcal{J}}\}$ has a total of $\mathcal{J}$ recurrent DAGs. A recurrent DAG G^j has a minimum inter arrival time (called the period) T^j and a relative constrained deadline D^j such that $T^j \geq D^j$ [3]. If the tasks of DAG G^j are released at time t, then they must complete their execution by $(t + D^j)$ and the next release happens no earlier than $(t + T^j)$. A DAG is modeled as: $G^j = (V^j, E^j)$, where V^j is the set of nodes (i.e., tasks) and $E^j \subseteq (V^j \times V^j)$ are the edges (i.e., dependencies) of the G^j.

A node $\tau^{i,j} \in V^j$, is a sequential task and G^j has a total of $\mathcal{I}^j$ sequential tasks. An edge represents a dependency between two tasks. If $(\tau^{p,j}, \tau^{q,j}) \in E^j$, then $\tau^{q,j}$ can start its execution only after $\tau^{p,j}$ finishes its execution. Without loss of generality, we assume that every DAG has one task with no incoming

edges, called source, and is denoted by $\tau^{src,j}$. We assume every DAG has one task with no outgoing edges, called a sink, and is denoted by $\tau^{sink,j}$.

A task $\tau^{i,j}$ has $\mathcal{P}$ different WCETs, one for each processor and denoted by $c_x^{i,j}$ for $x = 1 \ldots \mathcal{P}$. The WCET of a task is the same for all the processors of the same type. The minimum WCET of task $\tau^{i,j}$ among the $\mathcal{P}$ processors is: $c_{min}^{i,j} := \min_{x=1}^{\mathcal{P}}\{c_x^{i,j}\}$. We assume that all the tasks are allowed to be executed on all the processors, meaning that $c_x^{i,j} < \infty$ where $1 \leq x \leq \mathcal{P}$.

The *workload* of a task $\tau^{i,j}$ is the amount of computation that a task needs to complete. Regardless of the processor that a task is executed on, its workload is constant and equal to $c_{min}^{i,j}$. The workload of an entire DAG G^j is the sum of the workloads of all the tasks in G^j and is given by Eq. (1). The parameter W_1^j intuitively represents the sequential execution of one-by-one task where each such task executes on its best processor and thus takes its minimum WCET to finish execution.

Definition 1. *Workload of DAG G^j with $\mathcal{I}^j$ tasks:*

$$W_1^j := \sum_{i=1}^{\mathcal{I}^j} c_{min}^{i,j} \tag{1}$$

A source-to-sink path or simply path γ^j of DAG G^j is a chain of nodes that $\gamma^j = (\tau^{p,j}, \tau^{p+1,j}, \ldots, \tau^{q-1,j}, \tau^{q,j})$ where $(\tau^{i,j}, \tau^{i+1,j}) \in E$ such that $p \leq i < q$, $\tau^{p,j} = \tau^{src,j}$, and $\tau^{q,j} = \tau^{sink,j}$. All the different paths of G^j are denoted by set $paths^j$. The workload of an arbitrary path γ^j is the sum of the workload of the nodes that belong to that path and is given by:

$$\mathcal{W}(\gamma^j) := \sum_{\tau^{i,j} \in \gamma^j} c_{min}^{i,j}$$

The path with the largest workload among all the paths is called the *critical path* and is given by: $cp^j = \arg\max_{\gamma^j \in paths^j} \mathcal{W}(\gamma^j)$. The maximum workload of any path of G^j is given by Eq. (2).

Definition 2. *The maximum workload of any path in G^j:*

$$W_\infty^j = \mathcal{W}(cp^j) \tag{2}$$

The parameter W_∞^j intuitively represents the length of the schedule of the DAG on an infinite number of processors of each type. The total workload (Eq. (1)) and the workload of the critical path (Eq. (2)) can be computed in linear time in the representation of the DAG [10]. Even though the workload of a task is constant, the duration that it takes to execute it can vary depending on which processor the executes at run time (i.e., the task-processor mapping).

We define $\delta_x^{i,j}$ the speed that task $\tau^{i,j}$ can execute at the x^{th} processor for $x = 1 \ldots \mathcal{P}$ such that $\delta_x^{i,j} := c_{min}^{i,j}/c_x^{i,j}$. Let $\mathcal{O}_y^{i,j}$ be the y^{th} highest speed that task $\tau^{i,j}$ can execute on some processor. We specify the *speed-preference* of processors by a task $\tau^{i,j}$ using a sequence $\mathcal{O}^{i,j}$ as defined in Def. 3.

Definition 3 (Speed Preference). *We define* $\mathcal{O}^{i,j} = < \mathcal{O}^{i,j}_1, \mathcal{O}^{i,j}_2, \ldots \mathcal{O}^{i,j}_{\mathcal{P}} >$ *for* $\tau^{i,j}$ *as the sequence of non-increasing speeds where* $\mathcal{O}^{i,j}_1 \geq \mathcal{O}^{i,j}_2 \geq \ldots \geq \mathcal{O}^{i,j}_{\mathcal{P}}$.

For example, consider a platform with three processors and a task with WCETs $\{3, 1, 2\}$ for these three processors. The task has speeds $\{0.33, 1, 0.5\}$ and based on the Def. 3 the task has speed preference $\{1, 0.5, 0.33\}$.

3 Scheduler

Section 3.1 presents the algorithm of the proposed scheduler. Section 3.2 shows a property, called the greediness property, of the scheduler that is used for the formal worst-case analysis of the execution of the nodes of the sporadic DAGs.

Algorithm 1: $\mathcal{GUM}$

```
1 if Release or Completion event then
2     Check ReadyQ and RunQ for preemption, update platform's status.
3     while ∃τ^{i,j} ∈ RunQ can Migrate do
4         {τ^{curr}, sp_dst} = Find the task τ^{i,j} ∈ RunQ, with the smallest
            k ∈ O^{i,j} that ProcIndex(O^{i,j}_k) is idle
5         Execute τ^{curr} to ProcIndex(O^{curr}_{sp_dst}), update platform's status.
6     while ∃ idle and !ReadyQ.Empty do
7         τ^{i,j} = ReadyQ.head
8         Find smallest k ∈ O^{i,j} that ProcIndex(O^{i,j}_k) is idle
9         Execute τ^{i,j} to ProcIndex(O^{i,j}_k), update platform's status.
```

3.1 Algorithm of the scheduler

Alg. 1 presents the pseudocode of the $\mathcal{GUM}$ (Greedy, Unrelated, Maximum-speed-preference) scheduler. The scheduler is global (dynamic), work-conserving, migrative, and preemptive [3]. The scheduler's main design principle is to schedule a task to the idle processor with its highest speed-preference (Def. 3).

The scheduler is invoked in two cases: (i) when a DAG is released and its source is ready for execution, and (ii) a task completes its execution and its children are ready for execution (line 1). In both cases, the tasks are stored in the *ReadyQ* and are sorted based on their priorities. The *RunQ* keeps track of the tasks in execution based on their priorities. The highest and lowest priority tasks both in *ReadyQ* and *RunQ* are at the queue's head and tail, respectively.

The algorithm mainly has three different steps. In the *first* step, we check if there would be any **preemption** when all the processors are busy, and a task is awaiting execution with a relatively higher priority than that of a task in execution. We preempt the lowest priority among the executing tasks with the highest priority task in the *ReadyQ*. We update the status of the platform with the idle processors that are available after a preemption (line 2). In the *second* step, we check if there are currently executing tasks in *RunQ* that can **migrate** to a faster idle processor. We repeatedly in line 3-5 try to find a task from the *RunQ* with the highest speed-preference (which is the task with the

smallest index in the speed preference sequence) considering the processors that are currently idle. We migrate such a task to the faster idle processor and the processor it was executing on is marked as idle (i.e., status of the platform is updated in line 5). We check all the executing tasks if they can migrate, and if there is no task, we exit the loop in line 3-5. In the *third* step, we **dispatch** the tasks for execution from the *ReadyQ*. If there are idle processors and there are tasks ready for execution, we dispatch such a task to that idle processor on which it executes the fastest and we remove the task from *ReadyQ* (lines 6-9).

The migration cost is platform-dependent and such architectural characteristics of the platform are known during the WCET analysis which is an active research area [17]. We assume that the cost of migrations is already included in the WCET of the task.

3.2 Greediness property

In this section, we present a property of $\mathcal{GUM}$ called the *greediness property*. Based on this property, we will define the platform's minimum capacity that intuitively shows the minimum rate by which the application's workload can be executed on the platform. Before we present the property in Lemma 1, we introduce the notions of *scheduling point* and a stable time interval. A *scheduling point* is a point in time when $\mathcal{GUM}$ needs to schedule a task. Such a scheduling point is at the beginning of the schedule or when a task's release or completion occurs (Alg. 1). We call a time interval [a,b] a *stable time interval* when there is no scheduling point within the interval except at the endpoints in [a,b].

Lemma 1. ***(Greediness Property)****: If total p processors are busy executing some tasks during any stable time interval with the $\mathcal{GUM}$ scheduler where $p \leq \mathcal{P}$, then there is some task executing at least at its x^{th} fastest speed for $x = 1, 2, \ldots p$.*

The lemma can be proven by case analysis of the scheduled tasks. We need to consider three cases: (1) all processors are idle; (2.a) some tasks are still in execution, and no migration is possible; (2.b) some tasks are still in execution, and some tasks migrate to faster processors. The main idea of the proof of Lemma 1 is that if a total of $(x-1)$ processors are busy, then a new task from the ready queue in the worst-case would be dispatched for execution to a processor that provides at least its x^{th} fastest speed. Similarly, if x tasks are in execution, a task migrates to a new processor that has at least its x^{th} fastest speed. Lem. 1 essentially specifies the task-processor mapping that $\mathcal{GUM}$ can achieve in the worst case if p processors are busy where $p \leq \mathcal{P}$. Initially, it guarantees that some task executes during any stable time interval with at least its highest speed. Next, there is some task that executes with at least its second-highest speed and so on. Note that the greediness property is oblivious to the task's priority, and as a result, it holds for a broad range of priority-based schedulers.

Based on Lemma 1 that shows the worst-case task-processor mapping, we find in Eq. (3) the platform's minimum capacity that is always guaranteed to the applications. The inner minimum operation $\min_{i=1}^{\mathcal{I}^j}\{\mathcal{O}_x^{i,j}\}$ finds the minimum speed among the x^{th} speed-preference among all DAG tasks. The outer

summation $\sum_{x=1}^{\mathcal{P}}$ accumulates the minimum speeds for all the processors, i.e, for $x = 1 \dots \mathcal{P}$ processors.

Definition 4. *Minimum platform capacity:*

$$m_{\mathcal{P}}^{j} := \sum_{x=1}^{\mathcal{P}} \min_{i=1}^{\mathcal{I}^{j}} \{\mathcal{O}_{x}^{i,j}\} \tag{3}$$

One can analyze the platform capacity independent of the application for homogeneous and related multiprocessors [13,14] because each particular processor provides the same speed to all the tasks. However, for unrelated multiprocessors determinig the capacity is quite challenging as one needs to consider the WCETs of each task and its relationship to each particular processors. The minimum capacity in Eq. (3 intuitively shows the minimum rate at which an unrelated platform can execute the application's workload. Such an abstraction of the platform using minimum capacity in Eq. (3 is needed for timing analysis that we present in next section.

4 Makespan of a single DAG

In this section, we present the first part of the schedulability analysis, which is the intra-DAG analysis used for the response time computation. We analyze the worst-case schedule of an arbitrary single DAG, assuming it has dedicated access to the entire platform, and find the makespan of this single DAG.

To prove that the makespan is correct, we need to show that the $\mathcal{GUM}$ scheduler does not suffer from execution-time based timing anomalies, which occur when the DAG's schedule length becomes larger if a task of the DAG completes its execution earlier than its estimated WCET [13]. The work in [13,8] present an upper bound on the makespan for homogeneous processors by avoiding timing anomalies. In the rest of this section we present how we adapt the result form [13,8] for unrelated platforms.

In order to account for the worst-case schedulability analysis, we transform the original DAG into a different DAG, called the degraded DAG. The degraded DAG is constructed to capture the worst-case task processor mapping in the sense that the tasks may execute on their slowest processor, i.e., the tasks execute for their maximum WCET among the processor types. The advantage of this degradation is that the makespan of the degraded DAG can be computed based on previous work targeting homogeneous multiprocessors. Finally, we show that the DAG's makespan is essentially the makespan for the original DAG executed on the target unrelated multiprocessor platform. Definition 5 shows how we degrade the DAG such that we can apply results from earlier work in [13,8].

Definition 5. ***(DAG degradation)****: The DAG $G^{\hat{j}}$ is an isomorphic DAG to G^{j} ($V^{j} = V^{\hat{j}}$ and $E^{j} = E^{\hat{j}}$) that for every task of $G^{\hat{j}}$ instead of having $\mathcal{P}$ WCETs, the task has only one WCET denoted by ($c^{i,\hat{j}}$) and given as $c^{i,\hat{j}} := \max_{t=1}^{\mathcal{P}}\{c_{t}^{i,j}\}$, $\forall \tau^{i,j} \in V^{j}$. The total workload $W_{1}^{\hat{j}}$ and the workload of the critical path $W_{\infty}^{\hat{j}}$ of the degraded DAG $G^{\hat{j}}$ are computed by Eq. (1) and Eq. (2), respectively.*

Theorem 1. ***(Makespan)****: Given the degraded DAG G^j of G^j and an unrelated platform with a total of $\mathcal{P}$ processors, the following holds:*

1. *An upper bound on the makespan, denoted by $X^j_{\mathcal{P}}$, of G^j executed on a homogeneous platform with $\mathcal{P}$ processors is given by the right-hand side of Eq. (4):*

$$X^j_{\mathcal{P}} \leq \frac{W^j_1 + (\mathcal{P}-1) \cdot W^j_\infty}{\mathcal{P}} \tag{4}$$

2. *The upper bound on the makespan of G^j given by Eq. (4) is also an upper bound on the makespan of the G^j executed on the unrelated platform.*

Proof. We prove the two parts of the theorem as follows:

Part 1: By degrading the G^j to G^j, we pessimistically assume that all tasks execute for their maximum WCET. The platform with $\mathcal{P}$ processors that G^j is executing is a hypothetical homogeneous (WHH) platform. Consequently, the analysis for a single DAG on a homogeneous platform [13,8] can be applied and the makespan based on [13,8] is given in Eq. (4).

Part 2: During the execution of G^j on an unrelated platform, the execution time of a task can be *less* than its maximum WCET that we assume for G^j on the WHH due to two reasons. First, (a) the execution time shorter because of the pessimism in estimating the WCET [17]. Second, (b) a task of G^j on an unrelated platform can be mapped to a processor that ensures execution time that is less than its maximum WCET that we assume for the G^j when it is executed on the WHH. Eq. (4) that directly follows from [13,8] avoids the execution-time based timing anomalies and thus proved to be safe in [13,8]. Even if during the execution of G^j on an unrelated platform, some tasks because of (a) and/or (b) execute for less than their maximum WCET that we assume for G^j on WHH, the makespan in Eq. (4) is also the makespan of G^j for the unrelated platform. □

The makespan in Eq. 4 has quite broad applicability as it can be applied to any work-conserving scheduler like any fixed- or dynamic-priority scheduler.

5 Response time of a sporadic DAG set

The second part of the schedulability analysis is the inter-DAG analysis. We address the challenge to formally determine how long each DAG can interfere with the execution of another DAG under analysis. By combining the makespan analysis from the previous section with the interference analysis, we can compute the overall completion time (i.e., response time) for each of the sporadic DAGs, and we can check if all of them meet their deadlines or not.

The problem of computing the interference can be separated into two parts: (a) computing the maximum workload of each release of a DAG that can interfere with the other DAG, which is the main challenge for unrelated multiprocessors; and (b) finding the number of times a DAG can interfere with the other DAGs, which we can solve based on [2,5,15,3].

To address (a), we need to find an upper-bound on the workload that a DAG can interfere with the other DAGs. A DAG can be interfered by another DAG if there is no processor that the former can execute on. As a result, it is important to find the time duration that all other DAGs keep all the processors busy such that we can compute the total interference that all such DAGs can impose on a DAG under analysis. Please recall that the total workload W_1^j is defined based on the minimum among all the processors' WCET for each task. The total workload W_1^j can be seen as the sequential execution of a DAG that executes on a hypothetical ideal homogeneous platform with $\mathcal{P}$ processors.

A simple approach to find an upper-bound on the workload that a DAG can interfere with the other DAGs is the W_1^j from Def. 5. However, it would be pessimistic because we need to assume that all the tasks execute on their slower processor (i.e., for their maximum WCET). In Lem. 2 we introduce the inflated total workload that increases/inflates the total workload W_1^j such that it considers the heterogeneity of the platform and the worst-case task-processor mapping. To find the inflated workload of a DAG, we use the greediness property of $\mathcal{GUM}$ scheduler and the minimum capacity ($m_{\mathcal{P}}^j$) given in Eq. (3).

Lemma 2. *Let G^j is scheduled with $\mathcal{GUM}$ on an unrelated platform with $\mathcal{P}$ processors. The inflated total workload $w_{\mathcal{P}}^j$ upper bounds the workload W_1^j for an equivalent execution of G^j on a homogeneous platform with $\mathcal{P}$ processors, where:*

$$w_{\mathcal{P}}^j = \frac{W_1^j}{m_{\mathcal{P}}^j} \cdot \mathcal{P} \tag{5}$$

Proof. Consider a time interval during which all the processors are busy. During such a time interval, depending on which processors the tasks are executing on, the platform can offer different capacities to the application. First, by definition, the minimum capacity is the smallest and is used to lower-bound the platform's capacity for *any* time interval. As a result, the $W_1^j/m_{\mathcal{P}}^j$ is the longest time duration that all the processors are busy. Second, at any time interval, the platform's maximum capacity is $\mathcal{P}$ when each task executes with speed 1, which is equivalent to executing the tasks on a homogeneous platform where all the tasks execute for their minimum WCET. When all the processors are busy during any time interval, the workload can be consumed at a maximum rate of $\mathcal{P}$. The maximum inflated workload of a DAG assuming an unrelated platform with $\mathcal{P}$ processors is given by Eq. (5). □

The window analysis [2,5,15,3] for homogeneous multiprocessors finds the worst-case interference by considering the worst-case duration that the DAGs can keep all the processors busy and the number of times that each DAG can interfere each other DAG. By using the inflated workload (Eq. (5)) and the well-known window-based schedulability analysis framework [2,5,15,3], we compute the response time for unrelated multiprocessors. We use [5] to find the interfering workload for any work-conserving scheduler (`AWC`) that has the greediness property (Lem. 1). For `AWC`, the interfering workload is computed by considering that all the DAGs can interfere with the DAG under analysis (Th. 4 from [5]). For a

fixed-priority scheduling policy, e.g., rate monotonic `RM`, we need to consider only interference by the DAGs with higher priorities. For a dynamic-priority scheduler, e.g., earliest deadline first (`EDF`), the interference can be reduced by specializing the window analysis (Th. 5 from [5]). Finally, we combine the makespan from Section 4 with the worst-case interference from the window analysis to find the response time (Th. V.1 from [15]). For brevity, the exact equations form [5] to find the interference and the response time are not presented here. All the parameters used to find the response time can be computed in polynomial time and the response time can be computed in pseudo-polynomial time [3], which is acceptable for offline analysis of hard real-time applications [3].

6 Quantitative evaluation

We evaluate the response time computation based on synthetic DAGs and we check whether each DAG in a set of DAGs meets its deadlines or not. Section 6.1 presents our simulation framework, and Section 6.2 shows the simulation results.

6.1 Simulation framework

The platform has $\mathcal{P}$ processors and H processor types where $H \leq \mathcal{P}$. Each processor is assigned to an arbitrary processor type while ensuring that there is at least one processor of each of the H types. Parallel applications like OpenMP [11] can be modeled as a DAG, which has many tasks but has only a few unique task types [9]. Multiple tasks of a unique task type have the same functionality and thus have the same WCET on the same processor type. So we consider only the unique task types for generating synthetic DAGs.

We randomly generate between [1,10] unique task types for every DAG. Then, we generate the WCETs of the unique tasks that exist in every DAG. First, we determine for each task type the minimum WCET randomly between [1,100] and we associate a randomly chosen arbitrary processor type, called the initial processor type, to this WCET. Second for each unique software type, we add a randomly generated value between [1,$(H * 100)$] to the minimum WCET to generate the WCET for every processor of the platform ($\mathcal{P}$) that belongs to some processor type. When we add a new processor type different from the initial processor type, it will execute a task slower compared to the initial processor type. From the WCETs of the unique task types, we calculate the m_p^j from Eq. (3) and we consider only the unique task types because tasks that are of the same unique task type have the same WCET for all the processors.

We select a random value between [1,1000] to generate the total number of nodes ($\mathcal{I}^j$) for each DAG G^j. Then we randomly select a value between [$\mathcal{I}^j$,100000] as the value of W_1^j where each node is assumed to at least have an execution time of one time unit. For two given numbers $n \in (Z^+)$ and $b \in \mathcal{R}+$, the well-known `UUniFast` [7] algorithm can randomly generate an array of n numbers form an uniform distribution such that the sum of the numbers is exactly equal to b. Based on the `UUniFast` [7], we generate two arrays each with

the size equal to the number of unique task types (i.e., value of parameter a for `UUniFast`) and total value equal to the other parameter $b = 100\%$. The first array elements are the percentages of the workload that correspond to each unique task type. The second array elements are the percentages of workload for each unique task type that is part of the critical path that we use to compute the critical path's workload (W^j_∞).

Based on the $m^j_\mathcal{P}$ and W^j_1 we calculate the $w^j_\mathcal{P}$ given in Eq. (5). To compute the maximum total workload (W^j_1) and the maximum workload of the critical path (W^j_∞), initially we find the minimum speed among all the processors for each unique task type that we use to divide the workload and find the maximum workload. Next, based on the percentages of workload that correspond to each unique task type (first array), we find the workload that corresponds to each unique task type. Furthermore, based on the percentages of the workload of each unique task type that belongs to the critical path (second array), we find the workload of each unique task type that belongs to the critical path. By summing the workloads for each unique task type we find the (W^j_1) and (W^j_∞) that we use to compute the makespan ($X^j_\mathcal{P}$) given by Eq. (4).

The level of difficulty in scheduling some workload is generally controlled using a parameter called utilization: the higher is the utilization, the higher is the demand for computing resources, and consequently, the more difficult it is to schedule the workload without missing the deadline. We define utilization of G^j is $U^j = w^j_\mathcal{P}/T^j$. The total utilization of a set of total $\mathcal{J}$ DAGs is $U = \sum_j^\mathcal{J} U^j$. The period ($T^j$) of a DAG is selected randomly between the range [$X^j_\mathcal{P}$, $10 * w^j_\mathcal{P}$], similar to [15]. To generate DAGs with a specific utilization, called goal utilization, we first generate random DAGs. We add their utilization as long as the total utilization is smaller than the goal utilization. For the last DAG, we modify the period to fit the goal utilization. The relative deadline is randomly selected between [$X^j_\mathcal{P}$, T^j].

We calculate the response time where the DAGs are given priority based on rate monotonic (`RM`) and earliest deadline first (`EDF`). We also calculate the response time for an arbitrary priority ordering (`AWC`). In summary, the response time of the DAGs of each set of DAGs is computed using three different policies. All the tasks of the same DAG have the same priority.

All the randomly generated values that we use to create the synthetic DAGs are are uniformly random values. We compute the response time for 200 DAG sets at each utilization point, and we report the average percentage of DAG sets where all its DAGs meet their deadlines, called *acceptance ratio* —– the higher is the acceptance ratio, the better is the proposed analysis in guaranteeing the schedulability of parallel applications.

6.2 Results of sensitivity analysis

Figure 1 presents the simulations if we keep the number of processors fixed ($\mathcal{P} = 4$) and vary the utilization. The vertical axis is the acceptance ratio (%). The horizontal axis is the utilization that we change with step 0.25. Figure 2 shows the acceptance ratio if we keep the utilization fixed ($U = 2$) and we vary

the number of processors. The vertical axis is the acceptance ratio, and the horizontal axis is the number of processors $(1, 2, 4, \ldots, 128)$. For both figures, the left, center, and right graphs are the `AWC`, `EDF`, and `RM` scheduling policies, respectively. The different lines are the results for $H = \{1, 2, 3\}$.

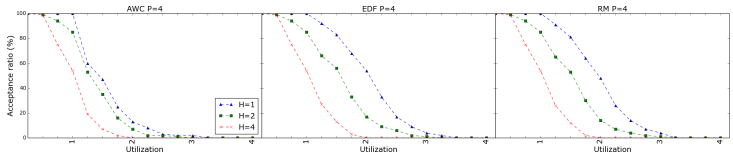


Fig. 1. Acceptance ratio for fixed number of processors and varying the utilization.

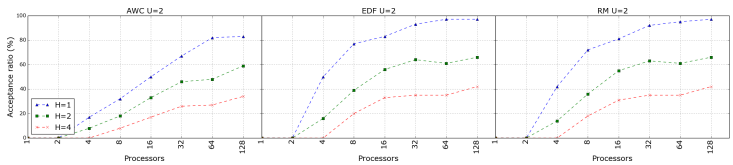


Fig. 2. Acceptance ratio for fixed utilization and varying the number of processors.

We can see that `EDF` and `RM` provide a higher acceptance ratio than `AWC`. The schedulability analysis of `EDF` and `RM` needs to consider the interference that the higher priority DAGs cause while `AWC` assumes that all the DAGs can interfere with another DAG. We observe that `EDF` and `RM` have similar acceptance rates. Next, by increasing the number of processor types (i.e., degree of heterogeneity), the acceptance ratio of $H = \{2, 4\}$ follows the same trend with $H = 1$. Also, by increasing H for each scheduling policies the acceptance ratio decreases. There are two reasons behind such a phenomenon. The first reason is that the added processor types are slower compared to the platform where $H = 1$. Please recall that if we increase H, we add a new, slower processor type compared to the case that $H = 1$ in our simulation framework. The platform with $H = 1$ can be seen as the best hypothetical homogeneous platform because all the tasks are executed for their minimum WCET in comparison to the heterogeneous platforms ($H = \{2, 4\}$). So it is expected that by increasing the number of processor types, we have a lower acceptance ratio compared to the case when all the tasks always execute at their minimum WCET. The second reason is the pessimism of the makespan computation. While the inter-DAG analysis takes advantage of the scheduler's greediness property, the makespan computation pessimistically assumes that all the tasks execute at their largest WCET.

Overall, we can apply the proposed response-time computation to a broad range of heterogeneous platforms, parallel applications, and scheduling policies. We are aware of the pessimism considered in our analysis in order to never

underestimate the completion time of an application in the worst case. Reducing the pessimism further but in a safe manner is an interesting future work.

7 Related work

There have been several works on determining the makespan of a single DAG on different types of multiprocessor platforms. The classic work by Graham [13] initiated a plethora of research for determining the makespan of parallel applications on homogeneous multiprocessors. The work [8] extended the analysis in [13] for Cilk-based parallel programs considering a work-stealing scheduler. The work [4] extends the result in [8] for related heterogeneous systems. None of these earlier works considers an unrelated platform that we consider in this paper. For *Typed* DAGs, each node can execute on exactly one type of processor [19,16]. The assumption that each node can execute only on one type of processor has limited applicability for emerging heterogeneous architectures available today (e.g., the big.LITTLE). Unlike [19,16], our work assumes that all the tasks can execute on all the processors. The [6] considers scheduling a set of independent sequential tasks on unrelated multiprocessors. Although these works consider quite a general processor model, the application model is quite restrictive; for example, sequential tasks cannot model dependencies among the tasks. For parallel independent tasks, the potential parallelism that we can achieve is limited because the intra-task parallelism is not allowed. The [14] proposes federated scheduling of sporadic DAGs on related heterogeneous multiprocessors. Since the related model is a special case of the unrelated model, our work is applicable not only in the context of [14] but also in other hardware and software setups. The problem of scheduling multiple DAGs on homogeneous multiprocessors is addressed in [15,18,12]. The homogeneous multiprocessor is a special case of an unrelated multiprocessor model. Our response time computation would be identical to [15] when applied to homogeneous multiprocessors, which shows the backward compatibility feature of our analysis.

In summary, earlier work on scheduling parallel applications on multiprocessors considered either a restrictive model of the application or a restrictive model of the hardware platform. However, a general setting like multiple recurrent DAGs executing at an unrelated heterogeneous platform can model many real-time applications requiring high computation power. Our work fills the gap by using the application, processor, and scheduler models that are very general as they can be applied to various parallel applications and hardware platforms.

8 Conclusion

This paper addresses the problem of computing the response time of sporadic DAGs on an unrelated heterogeneous multiprocessor platform. The $\mathcal{GUM}$ scheduler is proposed and analyzed to compute the makespan of a single DAG. Based on the makespan of single DAG and interference that one DAG can suffer from another DAGs, we developed a response time computation that is applicable

to any work-conserving scheduler with the greediness property. Simulations are presented to show the effectiveness of our proposed approach. To the best of our knowledge, this is the first work that addresses the problem of scheduling a set of constrained-deadline sporadic DAGs on unrelated machines. The main salient feature of the proposed research is its generality as it applies to a variety of hardware platforms, application models, schedulers.

References

1. ARM: The future of compute, re-imagined. `https://www.arm.com/why-arm/technologies/dynamiq` (2017)
2. Baker, T.P.: Multiprocessor edf and deadline monotonic schedulability analysis. In: RTSS. IEEE (2003)
3. Baruah, S., Bertogna, M., Buttazzo, G.: Multiprocessor scheduling for real-time systems. Springer (2015)
4. Bender, M.A., Rabin, M.O.: Scheduling cilk multithreaded parallel programs on processors of different speeds. In: SPAA (2000)
5. Bertogna, M., Cirinei, M.: Response-time analysis for globally scheduled symmetric multiprocessor platforms. In: RTSS. IEEE (2007)
6. Bertout, A., Goossens, J., Grolleau, E., Poczekajlo, X.: Workload assignment for global real-time scheduling on unrelated multicore platforms. In: RTNS (2020)
7. Bini, E., Buttazzo, G.C.: Biasing effects in schedulability measures. In: ECRTS. IEEE (2004)
8. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. JACM (1999)
9. Chronaki, K., et al.: Criticality-aware dynamic task scheduling for heterogeneous architectures. In: ACM, ICS (2015)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2009)
11. Duran, A., et al.: Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In: ICPP (2002)
12. Fonseca, J., Nelissen, G., Nélis, V.: Schedulability analysis of dag tasks with arbitrary deadlines under global fixed-priority scheduling. Real-Time Systems (2019)
13. Graham, R.L.: Bounds on multiprocessing timing anomalies. SIAM journal on Applied Mathematics (1969)
14. Jiang, X., Guan, N., Long, X., Yi, W.: Semi-federated scheduling of parallel real-time tasks on multiprocessors. IEEE RTSS (2017)
15. Melani, A.e.a.: Response-time analysis of conditional dag tasks in multiprocessor systems. In: ECRTS (2015)
16. Peng, X., Han, M., Deng, Q.: Response time analysis of typed dag tasks for g-fp scheduling. In: International Symposium on Dependable Software Engineering: Theories, Tools, and Applications. Springer (2019)
17. Reinhard, W., et al.: The worst-case execution-time problem overview of methods and survey of tools. ACM TECS (2008)
18. Ueter, N., von der Brüggen, G., Chen, J.J., Li, J., Agrawal, K.: Reservation-based federated scheduling for parallel real-time tasks. In: IEEE RTSS (2018)
19. Yang, K., Yang, M., Anderson, J.H.: Reducing response-time bounds for dag-based task systems on heterogeneous multicore platforms. In: ACM RTNS (2016)
20. Zahaf, H.E.e.a.: The hpc-dag task model for heterogeneous real-time systems. IEEE Transactions on Computers (2020)

Chapter 7

Paper IV

Federated scheduling of sporadic DAGs on unrelated multiprocessors

Petros Voudouris, Per Stenström, Risat Pathan

Under submission, 2021.

Federated scheduling of sporadic DAGs on unrelated multiprocessors

ANONYMOUS

This paper presents a federated scheduling algorithm for implicit-deadline sporadic DAGs that execute on an unrelated heterogeneous multiprocessor platform. We consider a global work-conserving scheduler to execute a single DAG exclusively on a subset of the unrelated processors. Formal schedulability analysis to find the makespan of a DAG on its dedicated subset of the processors is proposed. The problem of determining each subset of dedicated unrelated processors for each DAG such that the DAG meets its deadline (i.e., designing the federated scheduling algorithm) is tackled by proposing a novel processors-to-task assignment heuristic using a new concept called *processor value*. Empirical evaluation is presented to show the effectiveness of our approach.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: Federated, work-conserving, multiprocessor, scheduling, heterogeneous, unrelated, sporadic, DAGs

ACM Reference Format:

Anonymous. 2018. Federated scheduling of sporadic DAGs on unrelated multiprocessors. In *Proceedings of* . ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

New functions are continuously being added in many real-time systems to improve both the safety and the quality of service for end-users (e.g., autonomous vehicles, industrial robots). Integrating new functions requires high computing power, which one can achieve using special hardware. Heterogeneous multiprocessors are key to provide such computation power for many modern real-time applications [3, 4, 18, 21, 28, 29, 35, 36]. In addition, real-time applications that are implemented using parallel programming models like OpenMP [5] can also effectively exploit such parallel heterogeneous architectures. However, the main challenge to use a heterogeneous platform for real-time applications is to guarantee *time predictability*, i.e., how to ensure offline that the deadlines of the parallel applications will be met at runtime. This paper proposes the design and analysis of a scheduling algorithm for executing a collection of parallel real-time applications on a heterogeneous multiprocessor platform while guaranteeing timeliness.

The directed acyclic graph (DAG) is quite powerful to model many parallel real-time applications. A DAG models one application and has a set of nodes representing sequential tasks and a set of edges representing the dependencies among the tasks. We consider the problem of scheduling a set of sporadic DAGs with implicit deadlines on an *unrelated* heterogeneous multiprocessor platform. Note that the unrelated model is general enough to cover both the homogeneous and related (i.e., uniform [7]) multiprocessor models, and many commercial heterogeneous platforms are compliant with the unrelated model [3, 4]. One of the unique features of an unrelated platform is that the processors or cores of the platform are of *different types* [7]. Unlike the related machine model, different tasks of a DAG can have different speeds of execution on the same processor type of an unrelated platform. Exploring such speed relationships that the

Manuscript submitted to ACM

tasks of a DAG have to different processor types makes the design and timing analysis of a scheduling algorithm for unrelated platform really challenging.

The works in [1, 2, 8, 11, 31, 33] consider the scheduling of independent and sequential tasks on an unrelated platform. Unfortunately, sequential independent tasks neither can exploit intra-task parallelism nor can they be used to model, for example, data dependencies. Other works [26, 27, 30, 38, 42–44] on scheduling DAGs on unrelated platform assume that a node can only execute on exactly one type of processor (typed DAGs [26, 42, 43] and HPC-DAGs [44]) or is applicable only to related platforms [30]. In summary, earlier works on heterogeneous multiprocessors consider a restrictive model of the tasks or the platform. Our work bridges this gap by considering scheduling a set of DAGs where the nodes may have dependencies, and a node is allowed to execute on *any* processor type of an unrelated heterogeneous platform.

We consider the *federated scheduling* [6, 12, 19, 32, 41] of a collection of sporadic DAGs where each DAG is assigned a *dedicated* subset of the processors of an unrelated multiprocessor platform (called, a cluster). The advantage of federated scheduling is that no DAG in a cluster interferes with the execution of another DAG in a different cluster. However, the allocation of unrelated processors to the DAGs is more challenging than that of the related or homogeneous model because different tasks may have different speed relationships with each unrelated processor of the platform. Our proposed federated scheduling algorithm is designed by considering the complex speed relationships the tasks of a DAG have with the processors of the unrelated platform.

The problem of searching the clusters that would make all the DAGs meet their deadlines is equivalent to the classical bin-packing problem, which is NP-hard in the strong sense [23]. Therefore, finding an optimal allocation of the unrelated processors to all the DAGs under the federated scheduling paradigm is computationally intractable. To that end, the design of our federated scheduling algorithm uses a novel heuristic for processor allocation to the DAGs based on a concept called the *processor value*, which specifies how much overall benefit an unrelated processor offers to meet the deadline of a DAG. If each of the DAGs is assigned a cluster of processors, it is guaranteed that all the deadlines of all the DAGs will be met at runtime.

Whether a DAG meets its deadline on a given cluster or not depends on (i) the cluster-level scheduling algorithm, and (ii) the corresponding schedulability test. We consider global scheduling as the cluster-level scheduler — called the $\mathcal{GUM}$ scheduler — which is used to dispatch the nodes of a DAG on the processors of its cluster. The $\mathcal{GUM}$ scheduler has a special property, called *the greediness property*, that always tries to execute tasks on a relatively faster processor whenever such a processor becomes idle at runtime, for example, due to the completion of another task. We present the schedulability analysis of $\mathcal{GUM}$ scheduler based on the greediness property. The outcome of the analysis is a mathematical expression that can be used to compute an upper bound on the completion time (i.e., makespan) of a DAG on its cluster. Another salient feature of the analysis of $\mathcal{GUM}$ scheduler is that it is oblivious to the priorities of the tasks of a DAG. Consequently, our analysis is also applicable to a wide variety of cluster-level schedulers like fixed- or dynamic-priority-based scheduling algorithms.

We evaluate the effectiveness of our proposed federated scheduling algorithm using randomly generated DAGs. We perform intensive sensitivity analysis of our proposed test by varying different parameters of our assumed system model. We also compare the proposed schedulability test for federated scheduling with that of the state-of-the-art[1] in [30, 34]. Our proposed federated scheduling algorithm outperforms the state-of-the-art in terms of satisfying the timing constraints of randomly generated sets of sporadic DAGs.

This paper makes the following contributions.

[1]We adapted the state-of-the-art analysis to apply in the context of our assumed system model since we could not find any earlier work that considers a general system model as ours.

- This paper introduces the concept *processor value* to capture the complex speed relationships that the tasks of a DAG have with each processor of an unrelated heterogeneous platform.
- A federated scheduling algorithm is designed based on processor value to find the feasible clusters for a set of implicit-deadline sporadic DAGs. Each DAG executing on a dedicated cluster does not interfere with the execution of another DAG on a different cluster.
- We consider the $\mathcal{GUM}$ scheduler as the cluster-level scheduler, which is based on the global scheduling paradigm.
  - The schedulability analysis of $\mathcal{GUM}$ scheduler is presented to determine the makespan of a DAG on a given cluster. The makespan is used to verify whether the deadline of the DAG on the cluster is met or not.
  - The $\mathcal{GUM}$ scheduler is shown to possess the greediness property: a task always executes on a relatively faster processor whenever such a processor becomes idle during runtime. Executing the tasks on relatively faster processors makes the overall application finish early and thus is more likely to meet its hard deadline.
- Our empirical investigation shows that the proposed federated scheduling is effective for different variations of the system's parameters. The proposed schedulability test for federated scheduling outperforms the state-of-the-art in terms of guaranteeing the real-time constraints of randomly generated sets of sporadic DAGs.

Paper organization: Section 2 introduces the model for the platform and the application. Next, Section 3 introduces the scheduler that we use to schedule the tasks of a single DAG. Section 4 presents the makespan computation of a single DAG that executes in isolation on a set of unrelated processors. In Section 5 we propose, based on the processor value parameter, the federated scheduling algorithm that we use to assign the processors to the DAGs. Furthermore, In Section 6 we evaluate the proposed federated scheduling algorithm. First, we introduce the simulation framework, and then we evaluate the effectiveness of the federated scheduling algorithm for unrelated and related multiprocessors. Finally, Section 7 presents the related work before we conclude the paper in Section 8.

2 SYSTEM MODEL

This section presents the model of the platform and applications that we consider in this paper. A heterogeneous multiprocessor platform is modeled by a set of $\mathcal{P}$ unrelated processors. A processor has an index x such that $1 \leq x \leq \mathcal{P}$ and the set of processor indices is denoted by $\mathcal{M}$ such that $\mathcal{M} = \{1, 2, \dots \mathcal{P}\}$. In this paper, whenever we say "processor x", we refer to the processor with index x.

A sporadic DAG set with $\mathcal{N}$ DAGs is defined by $\Gamma = \{G^1, \dots G^{\mathcal{N}}\}$, where each G^j is a DAG with index $j \in \{1, 2, \dots, j, \dots, \mathcal{N}\}$ that models one parallel application. Each G^j is released repeatedly with a minimum inter-arrival time (called the period), denoted by T^j. The execution of each release of G^j needs to be completed by its relative deadline, denoted by D^j. We assume implicit deadlines and it holds that $T^j = D^j$.

A sporadic DAG G^j with $\mathcal{I}^j$ nodes is defined by $G^j = (V^j, E^j)$ where, set V^j contains the nodes of the DAG and $E^j \subseteq (V^j \times V^j)$ are the edges among the nodes. A node with index i of G^j is denoted by $\tau^{i,j} \in V^j$ where $1, \leq i \leq \mathcal{I}^j$ and such a node represents a sequential task. An edge represents a dependency between two nodes/tasks. If $(\tau^{p,j}, \tau^{q,j}) \in E^j$, then $\tau^{q,j}$ can start its execution only after $\tau^{p,j}$ finishes its execution. Without loss of generality, we assume that every DAG has one task with no incoming edges, called the *source* and is denoted by $\tau^{src,j}$. In addition, we assume every DAG has one task with no outgoing edges, called the *sink* and is denoted by $\tau^{sink,j}$.

Let $c_x^{i,j}$ denote the WCET of $\tau^{i,j}$ when it executes on processor x, where $x \in \mathcal{M}$. We assume that each task $\tau^{i,j}$ can execute on any processor and can finish the execution in a bounded time, i.e. $0 < c_x^{i,j} < \infty$. We denote $c_{min}^{i,j} := \min_{x \in \mathcal{M}}\{c_x^{i,j}\}$ as the minimum WCET of $\tau^{i,j}$ on any processor. In definition 2.1, we define the total workload of the

DAG, denoted by W_1^j, to capture the execution requirements of an application G^j such that the workload is the sum of the minimum WCET of each task. The parameter W_1^j intuitively is the length of the sequential schedule assuming the best task-processor mapping, i.e., all tasks are executed sequentially for their minimum WCET.

Definition 2.1. Total workload of G^j is defined as follows:

$$W_1^j := \sum_{i=1}^{I^j} c_{min}^{i,j} \tag{1}$$

A *source-to-sink path* or simply path γ^j of a DAG G^j is a chain of nodes that $\gamma^j = (\tau^{p,j}, \tau^{p+1,j}, \ldots, \tau^{q-1,j}, \tau^{q,j})$ where $(\tau^{i,j}, \tau^{i+1,j}) \in E$ such that $p \leq i < q$ and $\tau^{p,j} = \tau^{src,j}$ and $\tau^{q,j} = \tau^{sink,j}$. The set of all paths in G^j is denoted by $paths^j$. We define the workload of path γ^j, which belongs to G^j, as follows:

$$\mathcal{W}(\gamma^j) := \sum_{\tau^{i,j} \in \gamma^j} c_{min}^{i,j} \tag{2}$$

We define the critical path as the path with the largest workload and is given by $cp^j = \arg\max_{\gamma^j \in paths^j} \mathcal{W}(\gamma^j)$. Definition 2.2 introduces the workload of the critical path. The parameter W_∞^j intuitively is the length of the schedule assuming a hypothetical platform with an infinite number of processors of each type where all the tasks execute for their minimum WCET.

Definition 2.2. Workload of the critical path cp^j of G^j:

$$W_\infty^j = \mathcal{W}(cp^j) \tag{3}$$

It is important to note that regardless of the task-processor mapping of the tasks during the actual execution, the workload of a path remains constant because the tasks' minimum WCET determines it. However, the duration that it will take to execute the path workload depends on the task-processor mapping during the actual execution. As a result, the path with the largest workload is not necessarily the path that would require the longest time duration to finish execution. Fig. 1 presents an example that illustrates this situation.

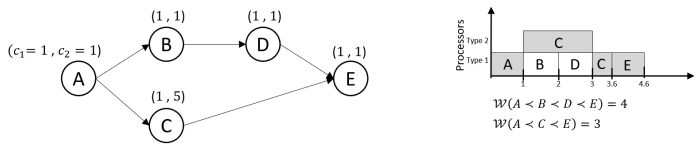


Fig. 1. Example of a path with the largest workload $A \prec B \prec D \prec E$ that it is not the path with the longest time duration. Note that task C migrates to a faster processor of type 1 when it becomes idle at time 3 (this greediness property will be explained shortly).

The left-hand side of Fig. 1 shows a DAG where each node has two different WCETs for two processors of different types. Based on Eq. (2), the path $A \prec B \prec D \prec E$ has workload four while the path $A \prec C \prec E$ has workload three. At the right-hand side of the figure, we assume that the tasks are scheduled in lexicographic order of the names of the tasks to a platform with one processor of type 1 and one processor of type 2. The example shows that path $A \prec C \prec E$ determines the schedule length, which has a smaller workload than $A \prec B \prec D \prec E$. We will shortly present how we can find the worst-case task-processor mapping that we can use together with the total workload and the critical path's workload to find the makespan of a single DAG.

The length of the sequential execution of G^j on a given single processor x where $x \in \mathcal{M}$ is given by: $S_x^j = \sum_{i=1}^{\mathcal{I}^j} c_x^{i,j}$. We define the single-processor utilization of G^j on processor x as follows:

Definition 2.3. The single processor utilization of G^j on a processor $x \in \mathcal{M}$ is:

$$U_x^j = \frac{S_x^j}{T^j} \tag{4}$$

The purpose of defining the utilization is to use the utilization-based schedulability test on a single processor where the tasks of the DAGs can execute sequentially and meet its deadline. Based on [6, 7, 9] the DAGs in Γ can meet their deadlines if the sum of the utilization (as per Definition 4) of the DAGs that are executing sequentially on a given processor x is smaller than one. More formally, if the Condition 2.1 is true, then the tasks in Γ will meet their deadlines during runtime by sequentially executing them with the earliest deadline first (EDF) scheduler on processor x.

Condition 2.1. *A single processor schedulability test of Γ on processor x assuming uniprocessor* EDF *scheduling:*

$$U_x^j \leq 1 - \sum_{G^k \in \Gamma, k \neq j} U_x^k, \forall G^j \in \Gamma \tag{5}$$

where a new DAG with utilization U_x^j can be assigned to processor x along with the other DAGs in set Γ.

Let $\delta_x^{i,j} := c_{min}^{i,j}/c_x^{i,j}$ denote the speed that $\tau^{i,j}$ has at processor x where $x \in \mathcal{M}$. The speed can take values between: $0 < \delta_x^{i,j} \leq 1$ and a higher value means faster execution. Please note that it is strictly larger than 0 because we have assumed that all the tasks can be executed on all the processors ($c_x^{i,j} < \infty, \forall x \in \mathcal{M}$). Let $O_x^{i,j}$ be the x^{th} highest speed that task $\tau^{i,j}$ can execute on *some* processor of $\mathcal{M}$ where $1 \leq x \leq \mathcal{P}$. We define the *speed-preference* of the processors for a task $\tau^{i,j}$ using a sequence $O^{i,j}$ as defined in Definition 2.4.

Definition 2.4. We define $O^{i,j} =< O_1^{i,j}, O_2^{i,j}, \ldots O_{\mathcal{P}}^{i,j} >$ of $\tau^{i,j}$, as the sequence of *non-increasing* speeds of the processors of $\mathcal{M}$ such that $O_1^{i,j} \geq O_2^{i,j} \geq \ldots \geq O_{\mathcal{P}}^{i,j}$.

A smaller index of the speed-preference sequence means a relatively higher speed for $\tau^{i,j}$. For example, assume that a platform has three processors and a task $\tau^{i,j}$ with WCETs $\{4, 1, 2\}$ on each processor. The task has speeds $\{0.25, 1, 0.5\}$ with speed-preference $\{1, 0.5, 0.25\}$. Given the speed-preference set, the second-highest speed for this example is $O_2^{i,j} = 0.5$. In the next section, the scheduler uses the speed-preference to select a relatively faster idle processor where a task may be executed.

3 SINGLE DAG SCHEDULER

This section introduces the scheduler that we use to schedule the tasks of a single DAG to an unrelated platform $\mathcal{M}$. Section 3.1 describes the algorithm of the scheduler. Section 3.2 shows a property of the scheduler that we will use to derive the makespan. We will use the scheduler and the schedulability analysis of this section later to assign each DAG to a dedicated subset of the processors and show the correctness of our proposed federated scheduling algorithm.

3.1 Description of the intra-DAG scheduler

This section presents the scheduler's pseudocode that we use to schedule the tasks of a DAG, called $\mathcal{GUM}$ (Greedy, Unrelated, Minimum-speed-preference-index). The $\mathcal{GUM}$ is global, meaning that all the tasks of G^j can execute on any processor of the unrelated platform $\mathcal{M}$. Although the makespan in this section is computed assuming all the $\mathcal{M}$ processors are available, we can apply the same analysis when we known the dedicated subset of the processors

assigned to a DAG. The scheduler is work-conserving, meaning that it never leaves a processor idle if there are ready tasks to execute. Alg. 1 presents the pseudocode of the scheduler.

The scheduler is invoked at the release of the source/root of G^j or at the completion of a task that belongs to G^j (line 1). In both cases, the tasks are stored in the *ReadyQ*, and the *RunQ* keeps track of the tasks in execution. The tasks in the *ReadyQ* are stored in an arbitrary order. Initially, we check if there are currently executing tasks in *RunQ* that can **migrate** to a faster idle processor. We repeatedly try to find a task from the *RunQ* with the highest speed-preference (which is the task that have in the speed-preference sequence the smallest index as is defined in Definition 2.4) considering only the currently idle processors. We migrate such a task to a faster idle processor, and the processor it was executing on is marked as idle.

We repeatedly check all the executing tasks if they can migrate, and if there is no task, we exit the loop in lines 2-4. Finally, we **dispatch** the tasks for execution from the *ReadyQ*. As long as there are idle processors and there are tasks ready for execution in the *ReadyQ* (lines 5-8), we select such a task from the head of the *ReadyQ* and dispatch the task to that idle processor on which it executes the fastest. We update the platform's status by marking the processor that the task is dispatched as busy, and we remove the task from *ReadyQ*.

Algorithm 1: $\mathcal{GUM}$

1 **if** *Release or Completion event* **then**
2 **while** $\exists \tau^{i,j} \in$ *RunQ can Migrate* **do**
3 $\{\tau^{curr}, sp\_dst\}$ = Find the task $\tau^{i,j} \in$ *RunQ*, with the smallest $k \in O^{i,j}$ that $ProcIndex(O^{i,j}_k)$ is idle
4 Execute τ^{curr} to $ProcIndex(O^{curr}_{sp\_dst})$, update platform's status.
5 **while** $\exists$ *idle and* !*ReadyQ.Empty* **do**
6 $\tau^{i,j}$ = *ReadyQ.pop*
7 Find smallest $k \in O^{i,j}$ that $ProcIndex(O^{i,j}_k)$ is idle
8 Execute $\tau^{i,j}$ to $ProcIndex(O^{i,j}_k)$, update platform's status.

3.2 Greediness property

We present the greediness property of the $\mathcal{GUM}$ scheduler that we will use to define the platform's minimum capacity that intuitively shows the minimum rate that the platform's processors can execute the application's workload. We will use the following definitions in Lemma 1. A scheduling point is a point in time when $\mathcal{GUM}$ needs to schedule a task. Such a scheduling point is at time zero or when the scheduler is invoked to schedule a task. We call a time interval $[a,b]$ a *stable time interval* when there is no scheduling point except at the endpoints in $[a,b]$.

Lemma 1. *(Greediness Property): If there is a total of* $0 < p \leq \mathcal{P}$ *busy processors executing some tasks during any stable time interval with the* $\mathcal{GUM}$ *scheduler, then there is some task executing at least at its* x^{th} *fastest speed for* $x = 1, 2, \ldots p$.

Proof. Let k, where $0 \leq k \leq p$, be the set of tasks that continue execution from one stable time interval to the immediately next stable time interval. In Alg. 1 the k tasks are stored in the *RunQ*. Also, assume that for some n where $0 \leq n \leq p - k$, there are n tasks that are newly scheduled at the beginning of the next stable time interval. Based on Alg. 1, the n tasks are stored in the *ReadyQ*. Let $k + n = p$, where $0 \leq p \leq \mathcal{P}$, be all tasks that we need to consider for execution in the new stable time interval. Let O^*_x denote the x^{th} fastest speed of some task. The asterisk (*) here is used

to specify an arbitrary task. We will prove this lemma by considering the two cases where the $\mathcal{GUM}$ scheduler takes scheduling decisions:

Case (1) - Migration: Let the task τ^{i_1,j_1} complete its execution at time a which is at the beginning of the stable time interval $[a,b]$. Let the task τ^{i_2,j_2} be among the k tasks that still continue execution during the stable time interval $[a,b]$. Please recall that the $\mathcal{GUM}$ scheduler first selects for migration the task that can execute on its most preferred processor, i.e., smallest index, in comparison to all the currently executing k tasks. Let task τ^{i_2,j_2}, if it migrates, has the most preferred processor among the k tasks that are still in execution. In the worst-case, τ^{i_1,j_1} was executing before its completion on a processor that is also for τ^{i_2,j_2} a faster processor. So the task τ^{i_2,j_2} can migrate to a processor that has at least $O_k^{i_2,j_2}$ speed because in the worst case there are $k-1$ tasks that can occupy the faster processors for τ^{i_2,j_2}. By following the $\mathcal{GUM}$ scheduler, the speeds at which tasks would start executing from the beginning of the stable interval in the worst case are $O_1^*, O_2^*, \dots, O_k^*$. This case covers the Alg. 1 lines 2-4.

Case (2) - Dispatch: The k already-executing tasks, either they continue executing on the processor on which they were executing on the previous stable interval, or they migrated to a faster processor. Because of case (1) the k tasks are executing at least with speeds $O_1^*, O_2^*, \dots, O_k^*$. Let the arbitrary task τ^* be the task that is selected for dispatching among the n new tasks that are ready for execution. The τ^* is going to execute at least with speed O_{k+1}^*, because in the worst-case all the k processors that provide higher speed $(O_1^*, O_2^*, \dots, O_k^*)$ for τ^* may be occupied by other tasks. Similarly, the remaining tasks from the n scheduled tasks are going to execute with speeds $O_{k+2}^*, \dots, O_{k+n}^*$. This case covers the Alg. 1, lines 5-8.

Therefore, based on cases (1) and (2), the $k+n=p$ tasks are executing, in the worst-case, with speeds O_x^*, for $x = 1, 2, \dots p$ where, $0 \leq p \leq \mathcal{P}$. □

The main idea of Lemma 1 is that if $x-1$ processors are busy, then a new scheduled task, in the worst case, would be executed on a processor that provides its x^{th} fastest speed because in the worst case, all the processors that provide higher speed are busy executing other tasks. Lemma 1 shows the worst-case task-processor mapping that the $\mathcal{GUM}$ scheduler can achieve because it guarantees that at any stable time interval that x tasks are executing, there is some task that executes with its highest speed. Next, there is some task that executes with its second-highest speed and so on. Finally, there is some task that executes with its x^{th} highest speed. By the worst-case, we mean that in another (better) case, the k^{th} dispatched task may also run on its highest speed processor if that processor has just become idle. In addition, because we proved the greediness property assuming arbitrary tasks, it holds for any priority ordering of tasks. Furthermore, the greediness property also holds if we extend the scheduler with a preemption mechanism. By selecting the removing the preempted task by a preempting task before the migration step, the greediness property holds because it is oblivious to the priority of the tasks.

Based on the greediness property in Lemma 1, we find in Eq. (6) the platform's minimum capacity to execute the application's workload when all the processors are busy. The inner minimum operation, $\min_{i=1}^{\mathcal{I}^j}\{O_x^{i,j}\}$ finds the minimum speed among the x^{th} speed-preference among all DAG tasks. The outer summation, $\sum_{x=1}^{\mathcal{P}}$, accumulates for $x = 1 \dots \mathcal{P}$ processors, the minimum speeds for all the processors.

Definition 3.1. Minimum capacity of the platform $\mathcal{M}$ with $\mathcal{P}$ unrelated processors for DAG G^j that is scheduled with $\mathcal{GUM}$ is given by:

$$m^j := \sum_{x=1}^{\mathcal{P}} \min_{i=1}^{\mathcal{I}^j}\{O_x^{i,j}\} \tag{6}$$

In Section 2, we define the workload (Eq. (1)) and the workload of the critical path (Eq. (3)) based on the minimum WCET of the tasks. In the following section, we will use the minimum capacity to inflate the workload such that we can compute the makespan by taking into account the dependencies among the tasks and the heterogeneity of the platform.

4 MAKESPAN OF A SINGLE DAG

This section presents the analysis to compute an upper bound on the worst-case schedule length, called the makespan, of a single DAG executed on an unrelated multiprocessor platform. A trivial makespan computation for unrelated multiprocessors can be developed by pessimistically assuming that all the tasks execute with their lowest speed, i.e., their maximum WCET among all the processors. Then we can find the makespan by [14, 24, 34] for scheduling DAGs on homogeneous processors because the execution assuming maximum WCET for all the tasks is equivalent to the homogeneous setup. Even though the estimation is safe, this approach could be quite pessimistic because we assume that all the tasks execute with their slowest speeds. We propose an alternative to finding a tighter makespan.

The greediness property of the $\mathcal{GUM}$ scheduler guarantees a relatively less pessimistic task-processor mapping. Please recall that if x processors are busy, then based on the greediness property, the worst-case task-processor mapping is the following: First, there is a task that executes with its highest speed. Next, there is some task which executes with its second-fastest speed. Finally, there is some task that executes with its x^{th} highest speed. The worst-case task-processor mapping is captured by the greediness property in the condensed parameter, i.e., the minimum capacity m^j in Eq. (6), that we use to find the makespan. Note that under the greediness property the tasks are not executing with their

A DAG's execution with a work-conserving scheduler can be separated into two parts: The time that processors execute the application's workload and the time that processors remain idle because of the tasks' dependencies or lack of enough parallelism. We introduce the total pseudo-workload to represent the processors' idleness, which intuitively shows the workload we could have executed if the idle processors were busy. Instead of considering all the task-processor mappings that can occur during run-time, we use the worst-case task processor mapping via the parameter m^j from Eq. (6) in order to inflate the workload in Lemmas 2 and 3. Finally, we use the scheduler's work-conserving property, and we find the makespan of a DAG by using the homogeneous multiprocessors formula [14, 24, 34].

Please recall that the total workload W_1^j is defined based on the minimum WCET of each task among all the processors. The total workload W_1^j can be seen as the sequential execution of a DAG that executes on a hypothetical homogeneous platform with $\mathcal{P}$ processors. The purpose of the inflated total workload is to increase/inflate the total workload W_1^j to include the platform's heterogeneity by considering the worst-case task processor mapping.

Lemma 2. *Let G^j be scheduled with $\mathcal{GUM}$ on a platform $\mathcal{M}$ with $\mathcal{P}$ unrelated processors. The inflated total-workload w^j upper bounds the total workload W_1^j for an equivalent execution on a homogeneous platform with $\mathcal{P}$ processors, where:*

$$w^j = \frac{W_1^j}{m^j} \cdot \mathcal{P} \tag{7}$$

Proof. Let a time interval be the duration of time where all the processors are busy. During any time interval, depending on which processors the tasks are executing on, the platform can have different capacities. By definition, the minimum capacity is the smallest and is used to lower-bound the platform's capacity at any time interval. As a result, the W_1^j/m^j is the longest time duration that all the processors are busy. During a time interval that all the processors are busy, the maximum capacity that the workload can be executed on is $\mathcal{P}$. The case that provides the maximum capacity is when all the tasks that are in execution during the time interval are mapped to processors that provide speed one, which is equivalent to executing the tasks on a homogeneous platform where all the tasks execute for their minimum

WCET. So, the maximum inflated workload of a DAG assuming a platform with $\mathcal{P}$ unrelated processors is given by Eq. (7). □

Because of the dependencies of the tasks, we have idle processors during the execution of the DAG. For a given stable time interval of the execution of G^j, let pseudo-workload be the product of the capacity of the idle processors and the time that they are idle. Let the *total pseudo-workload* be the sum of pseudo-workloads for all stable time intervals over the entire duration of the execution of the DAG, which shows the workload that the platform can execute if all idle processors were busy[2]. Because it may be computationally impractical to find the actual pseudo-workload, we compute an upper bound.

For a homogeneous platform, [25, 34], the largest workload of any path is W^j_∞, which executes at most on one processor, and the maximum capacity that can remain idle is $\mathcal{P} - 1$, so an upper bound on the pseudo-workload is $(\mathcal{P} - 1) \cdot W^j_\infty$. However, for an unrelated platform, the path with the largest workload does not necessarily take the longest time to execute because each task's execution time depends on the task-processor mapping. Please recall the example in Fig. 1. Similar to the inflated total workload in Lemma 2, we determine the total inflated pseudo-workload, denoted by $\bar{w}^j$, to cover the platform's heterogeneity as follows.

Lemma 3. *Let G^j be scheduled with $\mathcal{GUM}$ on an unrelated platform $\mathcal{M}$ that has $\mathcal{P}$ unrelated processors. The inflated total pseudo-workload $\bar{w}^j$ upper bounds the total pseudo-workload $(\mathcal{P}-1) \cdot W^j_\infty$ for an equivalent execution on a homogeneous platform with $\mathcal{P}$ processors, where:*

$$\bar{w}^j = \frac{(\mathcal{P} - 1) \cdot W^j_\infty}{m^j} \cdot \mathcal{P} \tag{8}$$

Proof. From [25, 34] we know for a homogeneous platform that the maximum idleness or the pseudo-workload is $W^j_\infty \cdot (\mathcal{P} - 1)$. It follows from Lemma 2 that the longest duration that is needed to execute the pseudo-workload by $\mathcal{P}$ processors is $(\mathcal{P} - 1) \cdot W^j_\infty / m^j$. An upper bound on the maximum capacity of the platform is $\mathcal{P}$. Consequently, the inflated pseudo-workload is calculated by Eq. (8). □

Lemmas 2 and 3 respectively inflate the total workload and the total pseudo-workload to resemble the execution of the application for a homogeneous platform with a total of $\mathcal{P}$ processors. We inflate these workloads to capture all the possible task-processor mappings that could occur during run time. In Theorem 1, we use the work-conserving property of $\mathcal{GUM}$ to find a safe upper bound of the makespan, denoted by X^j, for the original unrelated platform.

Theorem 1. ***(Makespan)**: An upper bound on the makespan of G^j scheduled with $\mathcal{GUM}$ on an unrelated platform with $\mathcal{P}$ processors is:*

$$X^j \leq \frac{W^j_1 + (\mathcal{P} - 1) \cdot W^j_\infty}{m^j} \tag{9}$$

Proof. Consider the execution of the tasks such that the length of the schedule is exactly equal to the makespan X^j (it will be evident that we do not need to know such an execution to prove this theorem). Let B be the actual total workload needed to execute the application, and let $\bar{B}$ be the actual total pseudo-workload (i.e., idle time) generated for the schedule with completion time X^j. Fig. 2 presents the two types of workloads as the area determined by the capacity of the platform (y-axis) and the execution time (x-axis). Because the scheduler is work-conserving, and it never

[2]Similar concept to *shadow threads* in [10] for a related (uniform) platform.

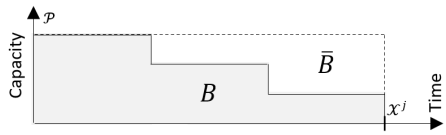


Fig. 2. Visualization of the makespan. The B area shows the application's actual total workload, and the $\bar{B}$ shows the actual total pseudo-workload.

leaves all the processors idle until the completion of the application, the makespan χ^j on a total of $\mathcal{P}$ homogeneous processors is equal to:

$$\chi^j \leq \frac{B + \bar{B}}{\mathcal{P}}$$

which is evident from Fig. 2.

To map the execution from the homogeneous to an equivalent execution on $\mathcal{P}$ unrelated processors, we upper bound the actual total workload and the actual pseudo-workload with the inflated workload and the inflated pseudo-workload, respectively. By Lemmas 2 and 3, it holds that $B \leq w^j$ and $\bar{B} \leq \bar{w}^j$ and we have:

$$\implies \chi^j \leq \frac{w^j + \bar{w}^j}{\mathcal{P}}$$

By replacing Eq. (7) and (8), we have:

$$\chi^j \leq \frac{W_1^j + (\mathcal{P} - 1) \cdot W_\infty^j}{\mathcal{P}} \cdot \frac{\mathcal{P}}{m^j}$$

which in trun is equivalent to:

$$\chi^j = \frac{W_1^j + (\mathcal{P} - 1) \cdot W_\infty^j}{m^j}$$

□

To theoretically evaluate the proposed makespan calculation, we compare it to the optimal-schedule-length (i.e., minimum completion time), denoted by OPT. Because it is intractable to find the optimal schedule length, we find a lower bound on the optimal schedule length as follows. Let an upgraded DAG be an isomorphic DAG to G^j ($V^j = V^{\hat{j}}$ and $E^j = E^{\hat{j}}$) that for every task instead of having $\mathcal{P}$ WCETs, each task has only one WCET which is the $c_{min}^{i,j}$. Because its task has one WCET, the analysis of DAG scheduling for homogeneous platform can be applied, and a lower bound on the optimal-schedule-length is found by $LB = \max\{W_\infty^j, \frac{W_1^j}{\mathcal{P}}\}$ for the upgraded DAG [14, 25] which is also a lower bound for the original DAG G^j.

Corollary 1. *The makespan is $\frac{2 \cdot \mathcal{P} - 1}{m^j}$ times larger than the optimal schedule length (OPT):*

Proof. From Eq. (9), given by Theorem (1), we have:

$$\chi^j = \frac{W_1^j + (\mathcal{P} - 1) \cdot W_\infty^j}{m^j}$$

$$\chi^j = \frac{\mathcal{P}}{m^j} \cdot \frac{W_1^j}{\mathcal{P}} + \frac{(\mathcal{P} - 1)}{m^j} \cdot W_\infty^j$$

By definition $LB \leq \texttt{OPT}$, so it holds that $\frac{W^j_1}{\mathcal{P}} \leq \texttt{OPT}$ and $W^j_\infty \leq \texttt{OPT}$.

$$\implies X^j \leq \frac{\mathcal{P}}{m^j} \cdot \texttt{OPT} + \frac{(\mathcal{P}-1)}{m^j} \cdot \texttt{OPT}$$

$$X^j \leq \frac{2 \cdot \mathcal{P} - 1}{m^j} \cdot \texttt{OPT}$$

□

The value $(\frac{2 \cdot \mathcal{P}-1}{m^j})$ is determined by the speeds of the tasks for the processors of the platform. If all the tasks of all DAGs have speed one for all the processors, then we have the homogeneous setup. The value (a.k.a approximation, speed-up, and resource augmentation) is equal to $(2 - \frac{1}{\mathcal{P}})$ which is equal to the classic bound derived by Graham [25].

We find the makespan by assuming that all the processors of set $\mathcal{M}$ are available for a DAG. In the following section, we will use the makespan in federated scheduling to test the schedulability of a DAG that is executing in isolation to a *subset* of unrelated processors that are available, called cluster. In the Corollary 2, we extend the notation of Theorem 1 with the cluster's index and we find the makespan of a DAG that executes at a cluster.

Corollary 2. *Let a cluster with index y, denoted by $\mathcal{K}_y$, be a subset of the processors in set $\mathcal{M}$. The cluster $\mathcal{K}_y$ has $\mathcal{P}_y$ unrelated processors ($|\mathcal{K}_y| = \mathcal{P}_y$). Without loss of generality, the processors are indexed by $x \in \{1, 2, \dots, x, \dots, \mathcal{P}_y\}$ and the $O^{i,j}_{x,y}$ represents the x^{th} fastest speed of the processors in $\mathcal{K}_y$ for task $\tau^{i,j}$. The upper bound on the makespan is computed based on Eq. (9) for a cluster $\mathcal{K}_y$ as follows:*

$$X^j_y \leq \frac{W^j_{1,y} + (\mathcal{P}_y - 1) \cdot W^j_{\infty,y}}{m^j_y} \tag{10}$$

where $W^j_{1,y}$, $W^j_{\infty,y}$, and m^j_y are computed by Eq. (1), Eq. (3), and Eq. (6) assuming that the only processors that are available for executing $\tau^{i,j}$ belong to $\mathcal{K}_y$. If the cluster has one processor ($\mathcal{P}_y = 1$), the utilization of that processor is denoted by $U^j_{1,y}$ and the Cond. 2.1 is used to check the schedulability of $\Gamma_y \subseteq \Gamma$.

After finding the makespan of a DAG that executes on a cluster, the next challenge is to determine the processors that compose the clusters. In the next section, we will describe how we select the processors to make the clusters and how we assign a DAG to a cluster such that all the DAGs of the sporadic DAG set meet their deadlines.

5 FEDERATED SCHEDULING

For federated scheduling, a DAG either gets a dedicated subset of processors (cluster) such that it can execute in isolation without being interfered with by other DAGs or is executed sequentially upon a single processor as in partitioned scheduling. Finding the optimal assignment of DAGs to processors for unrelated multiprocessors with respect to meeting all DAGs deadlines is an NP-hard problem [23]. So we have turned our focus on developing heuristics for the assignment of processors to the DAGs.

Because the processors are of different types, it is not clear what processors are good to select for each DAG such that all the DAGs would meet their deadlines. To characterize a processor with respect to meeting a DAG's deadline in Section 5.1 we introduce a concept, called *processor value*, that intuitively shows how "good" a processor is for a DAG concerning meeting its deadline. The processor value models the expected scheduling decisions that $\mathcal{GUM}$ would make during runtime by considering the speed-preference of all tasks of a DAG. We use the processor value in the federated scheduling algorithm to select suitable processors for each DAG. Finally, in Section 5.2 we present the

federated scheduling algorithm. The federated scheduling algorithm takes as input a sporadic DAG set and a set of unrelated processors. It returns `success` if it finds successful assignment of the processor to DAGs such that all the DAGs meet their deadlines otherwise returns `failure`.

5.1 Processor value

The processors of a heterogeneous platform have different capabilities to execute the workload of a task that belongs to a DAG. The speed-preferences for the processors characterize a task because they determine what processor the $\mathcal{GUM}$ would select during runtime. A DAG is composed of tasks with different speed-preferences for the processors. To determine if a processor can be beneficial for a DAG to meet its deadline, we introduce the processor value that intuitively shows how good is a processor for a DAG by taking into account the speed-preferences of each task that are used by $\mathcal{GUM}$ to schedule the tasks to a cluster.

Initially, at Definition 5.1 we introduce parameter $b^{i,j}_{s,x,y}$, that finds if the x^{th} processor that belongs to cluster $\mathcal{K}_y$, is the s^{th} highest speed for task τ^i that belongs to DAG G^j . We use the parameter $b^{i,j}_{s,x,y}$ to able to find the processor that provides the s^{th} highest speed for each task because each task would like to have processors that provide a higher speed of execution (i.e., smaller speed-preference) in order to complete its workload faster.

Definition 5.1. The boolean parameter is true if the processor $x \in \mathcal{K}_y$ is the s^{th} speed-preference of task $\tau^{i,j} \in G^j$, otherwise it is false:

$$b^{i,j}_{s,x,y} = \begin{cases} true & ProcIndex(O^{i,j}_{s,y}) = x \\ false & otherwise \end{cases} \tag{11}$$

where $O^{i,j}_{s,y}$ is the s^{th} fastest speed for $\tau^{i,j}$ in cluster $\mathcal{K}_y$ and $ProcIndex(O^{i,j}_{s,y})$ is the index of this processor in cluster $\mathcal{K}_y$.

Each DAG is composed of different tasks that have different speed-preferences for the processors. To be able to consider the speed-preferences of all the tasks that belong to a DAG, we use the parameter $b^{i,j}_{s,x,y}$ to count all the tasks that have the same speed on the same processor. More precisely, Definition 5.2 introduces parameter $cnt^j_{s,x,y}$ that counts the tasks that belong to DAG G^j and have processor $x \in \mathcal{K}_y$ as their s^{th} speed-preference.

Definition 5.2. The $cnt^j_{s,x,y}$ finds the number of tasks of G^j that have processor $x \in \mathcal{K}_y$ as the s^{th} speed-preference:

$$cnt^j_{s,x,y} = \sum_{i=1}^{T^j} b^{i,j}_{s,x,y} \tag{12}$$

In Definition 5.3 we introduce the processor value of a processor $x \in \mathcal{K}_y$ for DAG G^j. The $\sum_{s=1}^{\mathcal{P}_y}$ operation in Definition 5.3 accumulates for all the options the number of tasks that have processor $x \in \mathcal{K}_y$ as their s option. Because we assume that all the tasks can execute on all the processors, the number of options is equal to the number of processors of $\mathcal{K}_y$. The term $\frac{1}{s}$ in Eq. (13) introduces a weight for the $cnt^j_{s,x,y}$ to favor the processors that the $\mathcal{GUM}$ scheduler would try to select for the tasks of the G^j during runtime. A smaller value of s gives a larger weight, and the processors that provide higher speed get a higher value. The purpose of weight in determining the processor value of x is to capture the fact that a faster processor for the tasks is a very "valuable" or critical resource.

Definition 5.3. The processor value of processor $x \in \mathcal{K}_y$ with respect to G^j:

$$PV^j_{x,y} = \sum_{s=1}^{\mathcal{P}_y} \frac{1}{s} \cdot cnt^j_{s,x,y} \tag{13}$$

Because a DAG may need more than one processor to meet its deadline, we define the cluster value that accumulates the processor values of the processors that belong to $\mathcal{K}_y$, as follows: $\hat{\mathcal{P}}_y^j = \sum_{x \in y} PV_{x,y}^j$. In the next section, we will use the cluster value to find the suitable clusters of processors for each DAG in order to make all the DAGs schedulable. The purpose of defining cluster value is to select the (unassigned) DAG having the highest cluster value for the assignment of that cluster by considering the potentially feasible clusters of all the unassigned DAGs. Now, based on the cluster value, we next present our federated scheduling algorithm.

5.2 Description of the algorithm

The Alg. 2 presents the main functionality of the federated scheduling algorithm. We start by initializing the number of clusters ($\mathcal{Y}$) to zero (line 1). As long as there are unassigned DAGs from the sporadic DAG set Γ and there are available processors (lines 2 - 3), we try to assign the processors to the DAGs as follows. At line 4, we call Alg. 3 to find a temporary schedulable cluster (FTSC) for all the unassigned DAGs. The FTSC in Alg. 3 first computes the processor values of the available processors that have not been assigned to some DAG yet (*Avail*) with respect to DAG G. We rank/sort the processors based on their processor value (*Pref*). Based on the order of the processors specified by *Pref*, we build a temporary cluster (s) by adding one processor at each iteration of the loop at lines 5-10 in Alg. 3. We compute the makespan of G for the temporary cluster s. If the DAG is not schedulable, we add one more processor based on the *Pref* processor order to temporary cluster s, and we check if the DAG is schedulable. We continue adding processors until either the DAG is schedulable or we are out of processors, and we return zero (i.e., failure).

Among all the unassigned DAGs that we called the FTSC algorithm, we select the DAG (G^j) with the highest cluster value (*ClusterValue*, line 4). Next, at lines 5-8, we check if we can schedule the G^j to one of the already available clusters with a single processor. In case G^j is schedulable with single processor scheduling, we assign it to the available cluster with the smallest index that has one processor along with the other DAGs already assigned to this cluster, and we remove it from Γ. In case G^j is not schedulable to a single processor, we check if the FTSC in Alg. 3 found a schedulable cluster for G^j (line 10). If FTSC did not find a cluster that we can schedule G^j, then we return `failure` (line 16). Otherwise, we make a new cluster with the processors of (*ClusterProc*) of the temporary cluster, and we assign them permanently to G^j. The DAG is removed from Γ, and the cluster processors are removed by the $\mathcal{M}$ (lines 11-14). If all the DAGs of Γ are assigned, then we return `success` (line 19); otherwise, we return `failure` (lines 18).

6 EVALUATION

To quantitatively evaluate the federated scheduling algorithm, we generate synthetic DAG sets, and we check if all the DAGs of a DAG set meet their deadlines by varying different parameters of the system model. Section 6.1 introduces the simulation framework and Section 6.2 presents the simulation results.

6.1 Simulation framework

We model a platform by generating $\mathcal{P}$ processors. The platform has H processor types, where $H \leq \mathcal{P}$. An arbitrarily selected processor type characterizes each of the $\mathcal{P}$ processors, and we ensure that there is at least one processor of each of the H types. As a result, the number of processors of each processor type is randomly determined based on $\mathcal{P}$ and H.

We model a DAG by first generating the tasks that compose the DAG. Parallel applications like OpenMP [20] have many tasks but have only a few unique task types [17] — tasks that perform the same functionality but work on different data. Tasks that are of the same unique task type have the same functionality and thus have the same WCETs for the processors. For each DAG, we randomly generate between [1,10] unique tasks. For each unique task type, first, we

Algorithm 2: Is Γ Schedulable on $\mathcal{M}$

```
1  𝒴 = 0 // Total number of clusters
2  while Γ ≠ ∅ do
3      if M ≠ ∅ then
4          {ClusterProc, ClusterValue} = Call FTSC(G^j, M), ∀G^j ∈ Γ and select the G^j with the largest ClusterValue.
5          c = Find the first cluster index that P_c = 1 and for Γ_c ∪ G^j, the single processor schedulability test is true (Cor. 2 and
             Cond. 2.1 ), otherwise return 0.
6          if c ≠ 0 then
7              Γ_c = Γ_c ∪ G^j
8              Γ = Γ\G^j
9          else
               // Make new cluster
10             if ClusterValue ≠ 0 then
11                 𝒴 ++
12                 Γ_𝒴 = G^j
13                 Γ = Γ\G^j
14                 M = M\ClusterProc
15             else
16                 return failure
17     else
18         return failure
19 return success
```

Algorithm 3: FTSC(*DAG G*, *Cluster Avail*)

```
1  s = ∅
2  ClusterValue = MAX_FLOAT
3  List<float> Pref
4  Pref = Sort the Avail processors based on PV^j_{x,Avail}.
5  for p = 1; p < |Avail|; p++ do
6      Add the first p processors based on Pref to the temporary cluster s.
       // Check schedulability of G on cluster s based on Cor. 2
7      if X^G_s / D^G ≤ 1 and P̂^G_s ≤ ClusterValue then
8          ClusterValue = P̂^G_s
9          ClusterProc = s
10         break
11     s = ∅
12 if ClusterValue = MAX_FLOAT then
13     ClusterValue = 0
14 return {ClusterProc, ClusterValue}
```

determine its minimum WCET by randomly selecting a value between [1,100]. Then we randomly select a processor type to associate with the minimum WCET, and we call it the *initial* processor type. Next, for each unique task type and for each processor type, we add a randomly generated value between [1,($H * 100$)] to its minimum WCET to determine its WCET to the corresponding processor type. By adding a new processor type, we essentially add processors that are

slower compared to the initial processors that determine the minimum WCET. From the WCETs of the unique task types, we calculate the minimum capacity (m^j) from Eq. (6), and we consider only the unique task types because tasks that are of the same unique task type have the same WCET for all the processors.

After generating the WCET of the unique tasks for each processor of the platform, we generate the DAG. First, we generate the total number of nodes ($\mathcal{I}^j$) for each DAG G^j by selecting a random value between [1,1000]. Second, we generate the total workload (W_1^j) by selecting a random value between [$\mathcal{I}^j$,100000], where each node has at least one time unit of execution time. The `UUniFast` [13] algorithm, takes two numbers as input $n \in (\mathcal{Z}^+)$ and $b \in (\mathcal{R}^+)$ and randomly generates an array of size n, where each element of the array is selected from a uniform distribution such that the sum of elements of the array is exactly equal to b. We use the `UUniFast` algorithm to generate three arrays, each with the size equal to the number of unique task types and total value equal to 100%. Each element of the *first* array has a fraction of the total workload corresponding to a unique task type. Next, each element of the *second* array has a fraction of the unique task type's workload that belongs to the critical path. Finally, each element of the *third* array has a fraction of all the nodes that correspond to a unique task type. We use the first two arrays to find the workload of the critical path W_∞^j by summing all the unique task types workloads that belong to the critical path. We use the third array to compute the processor value $PV_{x,y}^j$ given by Eq. (5.3).

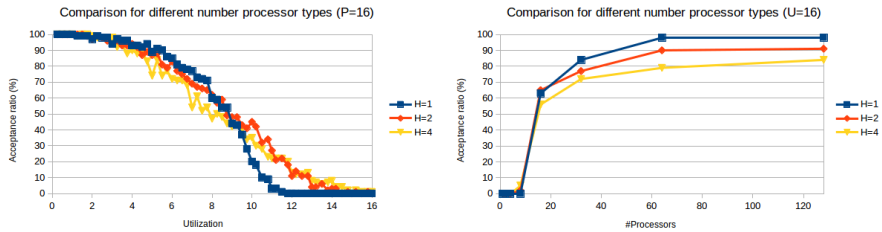
The utilization of the platform models the difficulty to successfully schedule a DAG set: a higher value of utilization means that it is more difficult to schedule a DAG set. Recall that by increasing the number of processor types, we always add slower processors. In order to compare a platform with $H = 1$ to another platform with $H > 1$, we define the utilization based on the inflated workload of a DAG for the unrelated platform under analysis. The inflated workload captures the heterogeneity of the platform. By defining the utilization of G^j on an unrelated platform as $U^j = w^j / T^j$, we can model the utilization of the platform by taking into account the fact that we have added slower processors (w^j is given by Eq. (7)). The total utilization of a DAG set with $\mathcal{N}$ DAGs is $U = \sum_{j=1}^{\mathcal{N}} U^j$. The period ($T^j$), which is equal to the deadline, of a DAG is selected randomly between the range $[X_{\mathcal{P}}^j, w^j]$, where $X_{\mathcal{P}}^j$ is the makespan of a DAG assuming that it has all the processors available. To generate a DAG set with a fixed utilization, called goal utilization, we first generate one-by-one random DAG, and we add their utilization. As long as the total utilization is smaller than the goal utilization, we continue generating random DAGs until a DAG has a larger utilization than the remaining utilization. For the last DAG, we modify the period to fit the goal utilization. All the randomly generated values that we use to create the synthetic DAGs are uniformly distributed. Based on the federated scheduling algorithm, we compute the schedulability of 200 DAG sets at each utilization point, and we report the average acceptance ratio — percentage of DAG sets that all their DAGs meet their deadlines.

6.2 Results

To quantitatively evaluate the federated scheduling algorithm, we perform three studies. First, we test the schedulability of the proposed federated scheduling by varying the number of processor types. Next, we compare our approach to global scheduling for unrelated multiprocessors. Finally, we specialize the system model for related multiprocessors, and we compare our approach to federated scheduling from previous works for related multiprocessors.

First, in Fig. 3a the vertical axis is the acceptance ratio, and the horizontal axis is the utilization. We keep the number of processors fixed to 16. The three lines are the simulation for multiprocessor platforms with $H = \{1, 2, 4\}$ number of processor types. Initially, we can see that for all the platforms, as the utilization increases, the acceptance ratio decreases. We can observe that for low utilization, the platform with $H = 1$ has the highest acceptance ratio. However,

as the utilization increases, we can see that the acceptance ratio of $H = \{2, 4\}$ is higher than $H = 1$. From the simulation framework, we know that when we add a new processor type, then we add a slower processor compared to the initial processor. We can imagine the initial processor as an "excellent" processor that can execute every task for its minimum WCET. We can assume that the initial processor would be extremely expensive. As a result, a multiprocessor platform with $H = 1$ would be a powerful platform but very expensive. By increasing the number of processor types, we add slower processors and sacrifice performance to reduce the platform's cost. In addition, because the utilization is defined by the inflated workload, the deadlines for the platforms $H = \{2, 4\}$ are longer compared to the platform with $H = 1$. To capture the heterogeneity of the platform, the inflated workload for a heterogeneous platform is larger compared to the case that $H = 1$ which is equal to the total workload of the application. We can argue that this experiment as offering platforms with different costs for longer deadlines that a client provides. From the simulation results, we see that we can use the platform $H = \{2, 4\}$ more efficiently than the platform $H = 1$. So, a client who can afford to have more extended deadlines can choose a platform with a lower cost having a relatively higher acceptance ratio.



(a) Fixed #Processors. (b) Fixed utilization.

Fig. 3. Different number of processor types.

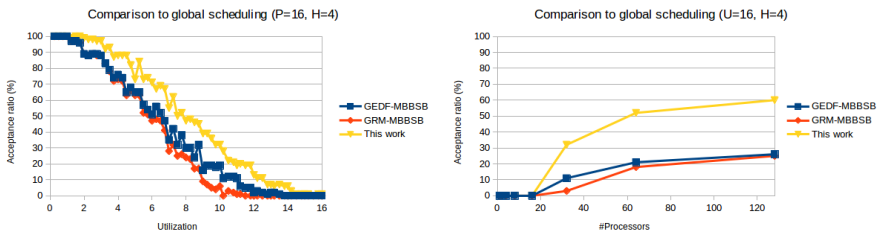
Second, in Fig. 3b the vertical axis is the acceptance ratio, and the horizontal axis is the number of processors, while we keep the utilization fixed to 16. Keeping the utilization fixed does not lead applications to have a relatively higher deadline for $H = \{2, 4\}$ in comparison to $H = 1$ as we have witnessed in the experiments presented in Fig. 3a. The platform with $H = 1$ achieves a higher acceptance ratio compare to the platforms with $H = \{2, 4\}$ because by increasing the number of processors, it is easier to find available processors for each DAG, and since all the tasks execute with speed, one, a high acceptance ratio is achieved. However, for $H = \{2, 4\}$ even though there are available processors, the processors are slower compared to $H = 1$ and achieves a lower acceptance ratio.

One of the main alternatives to federated scheduling is global scheduling. In global scheduling, all the DAGs share all the processors, which means that during runtime, a task of a DAG can execute on any idle processor. With global scheduling, the DAGs compete for the same processors, which introduces interference among the DAGs. The response time of a globally scheduled DAG is composed of the makespan of a DAG, assuming that all the processors are available and the maximum interference that the other DAGs introduce. There is no related work that proposes global scheduling for unrelated multiprocessors to the best of our knowledge. To compare the federated scheduling algorithm, we adapt for unrelated multiprocessors the global scheduling [34], originally developed for homogeneous multiprocessors. The adaptation is not trivial because the unrelated model is a generalization of the homogeneous model and requires to prove that we can find a safe estimation of the response time of DAGs that execute on unrelated multiprocessors. However, we do not address this problem in this paper, but we rather modify [34] only for the sake of comparison.

We compute the makespan given by Eq. (9) by assuming that all the processors are available for each DAG. In [34], to find the inter-DAG interference, the window analysis is used. The window analysis can be separated into two parts.

First, we need to find the maximum workload that a DAG can interfere with the other DAGs. To adapt the first part for unrelated multiprocessors, we replace the workload of a DAG with the inflated workload of a DAG given by Eq. (7). Second, the window analysis, by taking into account the relation of the periods and the deadlines of all DAGs, finds the maximum number of times a DAG can interfere with the other DAGs. For unrelated multiprocessors, we assume that the second part is the same because the platform does not influence the periods of the DAGs. By combining the two parts of the window analysis, we find the maximum interference that a DAG can experience by other DAGs. We extend our scheduler with preemption capabilities, and we use two well-known priority assignments, namely, rate monotonic (RM) and earliest deadline first (EDF). By assigning priorities to the DAGs allows us to consider only the higher priority DAGs to compute the interference that a DAG suffers. Finally, based on the makespan and the DAG's interference, we compute the response time. For global scheduling, the makespan of a DAG is shorter than federated scheduling since all the processors are assumed to be available when analyzing each task's makespan under global scheduling. However, we need to consider the interference among the DAGs in global scheduling. In contrast, for federated scheduling, there is no interference among the DAGs because the DAGs have dedicated processors, but the makespan is longer because a subset of processors is assigned to each DAG. Thus, it is not straightforward to conclude whether the global or federated scheduling dominates the other.

In Fig. 4a and 4b, we compare the federated scheduling algorithm to global scheduling. For Fig. 4a, the vertical axis is the acceptance ratio, and the horizontal axis is the utilization. We use a platform with 16 processors 4 processor types. In Fig. 4b the vertical axis is the acceptance ratio, and the horizontal axis is the number of processors. We keep the utilization fixed to 16. We compare this paper's federated scheduling algorithm to global RM denoted by GRM-MBBSB and global EDF marked by GEDF-MBBSB where "MBBSB" is from the initial letter of the authors' last name in [34].



(a) Fixed #Processors. (b) Fixed utilization.

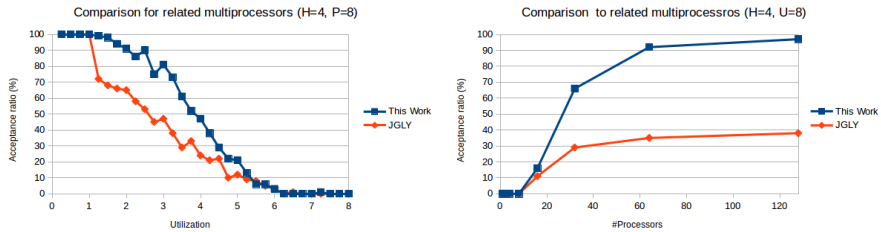
Fig. 4. Comparison to global scheduling.

In Fig. 4a and 4b it can be observed that federated scheduling achieves a higher acceptance ratio. For federated scheduling, the DAGs with an exclusive cluster to execute cannot interfere with the other DAGs. Also, DAGs that are scheduled sequentially can efficiently utilize the processors because the single processor EDF for implicit deadlines is optimal considering meeting the DAGs deadlines and can potentially fully use the processor. In contrast, for global scheduling, each DAG, even though it can exploit all the processors to minimize its makespan, interferes with all the lower priority DAGs and results in a lower acceptance ratio for these sets of simulations.

Finally, in Fig. 5a and 5b we adapt our system model for related multiprocessors. Because the related model is a special case of unrelated multiprocessors, the specialization is trivial by assuming that all the tasks of all the DAGs have the same speed-preferences for all the processors of the platform. Our single DAG scheduler becomes the same with the single DAG scheduler of [30], so our makespan is comparable to the makespan developed in [30]. For related multiprocessors, the makespan in [30] compared to Eq.9 is tighter because they model more precisely the pseudo-workload by exploiting

the characteristic of the related machines, which we cannot do since we consider unrelated machines. With the use of uniformity (Eq. (2) in [30]), they find a tighter estimation of the capacity idleness that is introduced by the dependencies among the tasks (i.e., smaller pseudo-workload) and, as a result, a tighter makespan. However, the main objective of [30] is to solve the underutilization for homogeneous multiprocessors. As a result, the processor to the DAG allocation mechanism is not aware of the different processor types' capabilities, which can lead to mapping the DAGs to processors that are incapable of meeting DAG's deadline.

In Fig. 5a the vertical axis is the acceptance ratio, and the horizontal axis is the utilization. We assume a platform with 8 processors and 4 processor types. In Fig. 5b the vertical axis is the acceptance ratio. The horizontal axis is the number of processors, and we assume a fixed utilization equal to 8, and we consider 4 processor types. We compare "this work" to [30], denoted by JGLY. It can be observed from both 5a, and 5b that the proposed federated scheduling algorithm can achieve a higher acceptance ratio compare to *JGLY*. Even though our approach has a less tight makespan compare to JGLY for related multiprocessors, it has a higher acceptance ratio because our processor to DAG allocation algorithm models the heterogeneity of the processors using the concept of processor value and the expected run-time scheduling decisions of the single DAG scheduler and assigns suitable processors for the DAGs such that all the DAGs can meet their deadlines.



(a) Fixed #Processors. (b) Fixed utilization.

Fig. 5. Specialization to related multiprocessors.

DAGs on unrelated multiprocessors. For the considered experiments, it is shown that the proposed federated scheduling algorithm outperforms global scheduling for unrelated multiprocessors. Finally, we see that the proposed approach is also effective for the related multiprocessors, which is a special case of unrelated multiprocessors.

7 RELATED WORK

There have been several works on determining the makespan of a single DAG on different multiprocessor platforms. The classic work by Graham [25] initiated much research for determining the makespan of parallel applications on homogeneous multiprocessors. The work in [14] extended [25] for the Cilk-based parallel program considering a work-stealing scheduler. The work in [10] extends the result in [14] for related heterogeneous systems. None of these earlier works considers an unrelated platform. Our work addresses this limitation by considering the makespan's computation for a single DAG on an unrelated platform. The works in [15, 39, 40] consider static scheduling of a single DAG on unrelated multiprocessors. However, static scheduling pre-assigns the tasks to the processors that may not fully utilize the platform like a work-conserving scheduler. Our work bridges this gap by designing a quite general work-conserving scheduler for unrelated platforms. For *Typed* DAGs, each node can execute on exactly one type of processor [26, 27, 38, 42, 43] and HPC-DAGs [44]. The assumption that each node can execute only on one type of processor has limited applicability for many heterogeneous architectures available today (e.g., the big.LITTLE).

Our work assumes that all the tasks can execute on all the processors. The problem of scheduling multiple DAGs on homogeneous multiprocessors is addressed for the global [16, 22, 32, 34, 37, 38] and federated [12, 30, 32, 41] scheduling algorithms. Since the homogeneous multiprocessor is a special case of an unrelated multiprocessor model, our results can also be applied to homogeneous multiprocessors. The work in [30] proposes makespan computation of a single DAG on a related heterogeneous machine. Then they apply this analysis to design a federated scheduling algorithm for multiple DAGs on homogeneous multiprocessors. The work in [30] — as the authors highlighted — can also be applied to schedule multiple DAGs on related heterogeneous machines. Since the related model is a special case of the unrelated model, our work can also be applied in the context of [30]. The works in [1, 2, 8, 11, 31, 33] consider scheduling a set of independent sequential tasks on unrelated multiprocessors. Although these works consider a general processor model, the application model is quite restrictive; for example, sequential tasks cannot model dependencies.

In summary, earlier works on scheduling parallel applications on multiprocessors make restrictive assumptions regarding the model of the application or model of the hardware platform. However, a general setting with sporadic DAGs scheduled on an unrelated heterogeneous platform can model many real-time practical applications requiring high computation power. Our work fills the literature gap by considering models of the application, processor, and scheduler that are very general because they can be applied to various parallel applications and hardware platforms.

8 CONCLUSION

This paper addresses the problem of providing timing guarantees for sporadic DAGs that execute on unrelated multiprocessors. We use a global scheduler to schedule the tasks of a single DAG on a dedicated cluster, and we perform the schedulability analysis to find the makespan of a single DAG on that cluster of unrelated processors. We introduce the concept of processor value to assign suitable processors to a DAG. Based on the processor value, we introduce a federated scheduling algorithm that assigns the processors to the DAGs such that all the DAGs meet their deadlines. Finally, the simulation results based on synthetic DAGs show that the proposed approach can achieve a higher acceptance ratio than global scheduling for unrelated multiprocessors and compare to the previous works on federated scheduling on related multiprocessors. Designing new heuristics based on processor value to determine feasible clusters under the federated scheduling paradigm is an interesting future work.

REFERENCES

[1] Björn Andersson and Gurulingesh Raravi. 2014. Real-time scheduling with resource sharing on heterogeneous multiprocessors. *Real-Time Systems* 50, 2 (2014), 270–314.

[2] Björn Andersson and Gurulingesh Raravi. 2016. Scheduling constrained-deadline parallel tasks on two-type heterogeneous multiprocessors. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. Association for Computing Machinery, Brest, France, 247–256.

[3] ARM. 2011. big.LITTLE Technology: The Future of Mobile. White paper, https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf.

[4] ARM. 2017. The Future of Compute, Re-imagined. https://www.arm.com/why-arm/technologies/dynamiq.

[5] Eduard Ayguadé et al. 2010. Extending OpenMP to survive the heterogeneous multi-core era. *International Journal of Parallel Programming* 38, 5 (2010), 440–459.

[6] Sanjoy Baruah. 2015. Federated scheduling of sporadic DAG task systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, IEEE, Hyderabad, India, 179–186.

[7] Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. 2015. *Multiprocessor scheduling for real-time systems*. Springer, .

[8] Sanjoy K Baruah, Vincenzo Bonifaci, Renato Bruni, and Alberto Marchetti-Spaccamela. 2019. ILP models for the allocation of recurrent workloads upon heterogeneous multiprocessors. *Journal of Scheduling* 22, 2 (2019), 195–209.

[9] Sanjoy K Baruah, Aloysius K Mok, and Louis E Rosier. 1990. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *[1990] Proceedings 11th Real-Time Systems Symposium*. IEEE, IEEE, Lake Buena Vista, Florida, USA, 182–190.

[10] Michael A Bender and Michael O Rabin. 2000. Scheduling Cilk multithreaded parallel programs on processors of different speeds. In *SPAA*.

[11] Antoine Bertout, Joël Goossens, Emmanuel Grolleau, and Xavier Poczekajlo. 2020. Workload assignment for global real-time scheduling on unrelated multicore platforms. In *RTNS*.

[12] Ashikahmed Bhuiyan, Zhishan Guo, Abusayeed Saifullah, Nan Guan, and Haoyi Xiong. 2018. Energy-efficient real-time scheduling of DAG tasks. *ACM TECS* (2018).

[13] Enrico Bini and Giorgio C Buttazzo. 2004. Biasing effects in schedulability measures. In *ECRTS*. IEEE.

[14] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *JACM* (1999).

[15] Tracy D Braun et al. 2001. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *JPDC* (2001).

[16] Peng Chen, Weichen Liu, Xu Jiang, Qingqiang He, and Nan Guan. 2019. Timing-Anomaly Free Dynamic Scheduling of Conditional DAG Tasks on Multi-Core Systems. *ACM TECS* (2019).

[17] Kallia Chronaki et al. 2015. Criticality-aware dynamic task scheduling for heterogeneous architectures. In *ACM, ICS*.

[18] Kallia Chronaki, Miquel Moretó, Marc Casas, Alejandro Rico, Rosa M Badia, Eduard Ayguadé, and Mateo Valero. 2019. On the maturity of parallel applications for asymmetric multi-core processors. *J. Parallel and Distrib. Comput.* (2019).

[19] Son Dinh, Christopher Gill, and Kunal Agrawal. 2020. Efficient Deterministic Federated Scheduling for Parallel Real-Time Tasks. In *RTCSA*. IEEE.

[20] Alejandro Duran et al. 2002. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *ICPP*.

[21] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *IEEE ISCA*.

[22] José Fonseca, Geoffrey Nelissen, and Vincent Nélis. 2019. Schedulability analysis of DAG tasks with arbitrary deadlines under global fixed-priority scheduling. *Real-Time Systems* (2019).

[23] Michael R Garey and David S Johnson. 1979. Computers and intractability. *A Guide to the* (1979).

[24] Ronald L Graham. 1966. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal* (1966).

[25] Ronald L. Graham. 1969. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics* (1969).

[26] Meiling Han, Nan Guan, Jinghao Sun, Qingqiang He, Qingxu Deng, and Weichen Liu. 2019. Response Time Bounds for Typed DAG Parallel Tasks on Heterogeneous Multi-cores. *IEEE TPDS* (2019).

[27] Meiling Han, Tianyu Zhang, Yuhan Lin, and Qingxu Deng. 2020. Federated scheduling for Typed DAG tasks scheduling analysis on heterogeneous multi-cores. *Journal of Systems Architecture* (2020).

[28] John L Hennessy and David A Patterson. 2017. *Computer architecture a quantitative approach, Sixth edition, Chapter 7*. Morgan Kaufmann.

[29] John L Hennessy and David A Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* (2019).

[30] Xu Jiang, Nan Guan, Xiang Long, and Wang Yi. 2017. Semi-Federated Scheduling of Parallel Real-Time Tasks on Multiprocessors. *IEEE RTSS* (2017).

[31] Eugene L Lawler and Jacques Labetoulle. 1978. On preemptive scheduling of unrelated parallel processors by linear programming. *JACM* (1978).

[32] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. 2014. Analysis of federated and global scheduling for parallel real-time tasks. In *IEEE ECRTS*.

[33] Alberto Marchetti-Spaccamela, Cyriel Rutten, Suzanne Van Der Ster, and Andreas Wiese. 2015. Assigning sporadic tasks to unrelated machines. *Mathematical Programming* (2015).

[34] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C Buttazzo. 2015. Response-time analysis of conditional DAG tasks in multiprocessor systems. In *ECRTS*.

[35] Alba Melo, Jesus Carretero, Per Stenstrom, Sanjay Ranka, and Eduard Ayguade. 2019. Trends on heterogeneous and innovative hardware and software systems.

[36] NVIDIA. 2018. Nvidia Jetson AGX Xavier and the new era of autonomous machines. Nvidia presentation, http://info.nvidia.com/rs/156-OFN-742/images/Jetson_AGX_Xavier_New_Era_Autonomous_Machines.pdf.

[37] Risat Pathan, Petros Voudouris, and Per Stenström. 2018. Scheduling Parallel Real-Time Recurrent Tasks on Multicore Platforms. *IEEE TPDS* (2018).

[38] Xuemei Peng, Meiling Han, and Qingxu Deng. 2019. Response Time Analysis of Typed DAG Tasks for G-FP Scheduling. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*. Springer.

[39] Gilbert C Sih and Edward A Lee. 1993. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE TPDS* (1993).

[40] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE TPDS* (2002).

[41] Niklas Ueter, Georg von der Brüggen, Jian-Jia Chen, Jing Li, and Kunal Agrawal. 2018. Reservation-based federated scheduling for parallel real-time tasks. In *IEEE RTSS*.

[42] Kecheng Yang and James H Anderson. 2014. Optimal GEDF-based schedulers that allow intra-task parallelism on heterogeneous multiprocessors. In *IEEE ESTIMedia*.

[43] Kecheng Yang, Ming Yang, and James H Anderson. 2016. Reducing response-time bounds for dag-based task systems on heterogeneous multicore platforms. In *ACM RTNS*.

[44] Houssam-Eddine Zahaf, Zahaf Houssam-Eddine, Nicola Capodieci, Roberto Cavicchioli, Giuseppe Lipari, and Marko Bertogna. 2020. The HPC-DAG Task Model for Heterogeneous Real-Time Systems. *IEEE Trans. Comput.* (2020).