Robust multiprocessor scheduling of industrial-scale mechatronic control systems

Robust Multiprocessor Scheduling of Industrial-Scale Mechatronic Control Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op maandag 04 juli 2016 om 16:00 uur

door

Shreya Adyanthaya

geboren te Mangalore, Karnataka, India

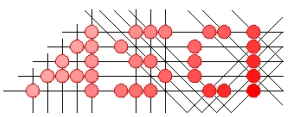
Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr.ir. A.B. Smolders
1^{e} promotor:	prof.dr.ir. A.A. Basten
copromotor:	dr.ir. M.C.W. Geilen
copromotor:	dr.ir. J.P.M. Voeten
leden:	prof.dr.sc. S. Chakraborty (Technische Universität München)
	prof.dr. B.F. Heidergott (Vrije Universiteit Amsterdam)
	prof.dr. J.J. Lukkien
adviseur:	dr.ir. R.R.H. Schiffelers (ASML)

Het onderzoek of ontwerp dat in dit proefschrift / proefontwerp wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Robust Multiprocessor Scheduling of Industrial-Scale Mechatronic Control Systems

Shreya Adyanthaya



This work was carried out in the ASCI graduate school.
ASCI dissertation series number 349.

Cover design by Ernest Mithun Xavier Lobo

Printed by CPI-Koninklijke Wöhrmann - The Netherlands

A catalogue record is available from the Eindhoven University of Technology Library. ISBN: 978-90-386-4074-4

Acknowledgements

"Vidya dadati vinayam, vinayadyati patratam,
patratvaddhanamapnoti, dhanaddharmam tatah sukham." *-Hitopadesha*
Translation: Education gives humility, humility gives worthiness, from worthiness one gets wealth, with wealth one does good deeds, from good deeds one gets joy.

The joy that I feel today as I am completing my thesis is the outcome of not just my own efforts but the support of many people who have played indispensable roles in helping me through these years. The least I can do is thank them.

Firstly, I must thank my promoter prof.dr.ir. Twan Basten, co-promoters dr.ir. Marc Geilen and dr.ir. Jeroen Voeten, and advisor dr.ir. Ramon Schiffelers for giving me the opportunity to work in the CARM 2G project that gave me ample exposure to both academia and industry. Most importantly it gave me the chance to see my work landed in practice. I am grateful for the crucial Friday morning discussions with Twan and Marc during which a lot of the research work of this thesis originated. I am grateful to Twan for always pulling me out of the details and helping me look at the bigger picture. His valuable guidance enabled me to ensure that the chapters of the thesis are well connected to form one coherent story. Despite his very busy schedule being the chairperson of the ES group, he still kept continuous track of my work and guided me during our discussions. I am very grateful to Marc for the detailed guidance that led in one way or another to the majority of the contributions of this thesis. I have always admired Marc's keen eye for detail and his humble and friendly nature. His constructive feedback helped me grow both in terms of research work as well as writing skills. Thank you for always finding time for discussions irrespective of your schedule and sometimes even after 18:00 hrs. I must thank Jeroen for all the white board discussions on how to formally approach the ASML's challenges that taught me innovative ways of doing applied research. Jeroen is the quintessential connection between the ES group, ASML and TNO-ESI and to me has been a motivational figure who has always given me reasons to feel proud of my work. I must also give credit to Jeroen's excellent presentation skills for teaching me the art of making and giving good presentations. I am thankful to Ramon for laying the foundation for CARM2G and for all the brainstorming sessions that led to the important scheduling achievements for ASML. Thank you for constantly telling me to plan my work which in turn helped me achieve my targets on time.

I extend my gratitude to prof.dr. Bernd Heidergott, prof.dr. Johan Lukkien, prof.dr.sc. Samarjit Chakraborty and prof.dr.ir. Bart Smolders for agreeing to be a part of my defense committee and for their time and effort in reviewing this manuscript. I specially thank Bernd for the insightful discussions at the VU

Amsterdam that led to interesting directions for future work and collaborations.

This research work was funded by ASML and TNO-ESI. I would like to thank Ramon Schiffelers, Jeroen Voeten, Nikola Gidalov, Pieter Nagelmaeker, Tom Engels, Jan van de Ven, Mark Flesch and Wilbert Alberts for all the important inputs that helped in the understanding of the scheduling challenges faced by ASML. Additionally, I thank Cees van Huët, Robert Hoogland, Elmar Beuzenberg, Jan Stoeten, Harry Borggreve and Jos Benschop for their managerial and financial support. My special thanks to Rolf Theunissen and Yuri Blankenstein from Altran for the many ASML discussions and for ensuring that the scheduling work was well integrated into the ASML workflow. I also thank Jan Schuddemat, Bart Theelen, Hristina Moneva, Boudewijn Haverkort, Reinier van Eck and Frans Beenker for their support from TNO-ESI. I specially thank Bart Theelen for the valuable inputs that led to the contributions of Chapter 2. Additionally, I would like to thank my master students at ASML, Santhosh Vaiyapuri, Zhihui Zhang and Beril Ok, for their work that directly or indirectly aided the research. Many thanks to my ASML PhD colleagues Raymond Frijns for the combined work on contention analysis, as well as João Bastos and Bram van der Sanden for the many cheerful chats at ASML.

I extend my heartfelt gratitude to all the members of the ES group for always being warm and friendly towards me. In particular, I would like to thank Martijn Koedam for helping me restore all the data that I had lost during an unfortunate laptop 'and' hard-disk crash recently. I also thank Sander, Marc and Martijn for the friendly atmosphere while sharing the office at floor 3. Special thanks to Marja, Rian and Margot for ensuring that all our activities were well planned and maintaining a warm and welcoming atmosphere in the group at all times. Thanks to Manil, Rehan, João, Francesco, Joost, Umar, Shubendu, Bram, Amir, Hadi and Róbinson for going from being colleagues to friends. I take with me fond memories of lunches shared, conferences attended together, games played and of course the yearly ES day celebrations.

I want to thank my close friends who made life in Eindhoven filled with fun and adventure. Thanks to my TU/e lunch partners Alok, Deb and Ram for the many de-stressing discussions on work, life and everything under the sun. I particularly thank Ajith and Shravan for helping me out at various points of my work. Thank you Saba and Mehaal for tolerating me during the stressful parts of my PhD and cheering me up when needed. Thank you Arnica, Ashu, Chetna, Geeta, Harshi, Hilda, Madhu, Shruthi, Shruti, Varshi, Abhinav, Adi, Amit, Ashwin, Guru, Hari, Mac, Manju, Poojith, Vikram, Vinod and Vivek for all our fun times together.

None of the above work could have been possible without the blessings of my parents and love from my sister, Shreshta. They supported me in my decision to do a PhD and always showered me with their unconditional love, which has always been my greatest pillar of strength. Finally and very importantly, I want to thank my loving fiancé Ernest for always believing in me and bringing the best out in me. Thank you for always being by my side, for caring for me in every way possible and for telling me, time and again, that I can definitely achieve this.

Summary

Robust Multiprocessor Scheduling of Industrial-Scale Mechatronic Control Systems

Over the last decades, embedded platform architectures for mechatronic systems have been composed of general-purpose components to benefit from their steady increase in average performance, most notably due to their increased processor clock frequencies. This trend has come to an end due to IC power dissipation limits. To meet future application demands, a shift has to be made to higher-parallel execution platforms. Given the control application specification one has to determine what application functionalities will run on which resources and in what sequence. This should be done in such a way that all timing requirements of all applications are met. The work of this thesis is motivated by the scheduling challenges faced by ASML, the world's leading provider of wafer scanners. A wafer scanner is a complex cyber-physical system that manipulates silicon wafers with extreme accuracy at high throughput. Typical control applications of the wafer scanner consist of thousands of precedence-constrained tasks with latency requirements. Machines are customized so that precise characteristics of the control applications to be scheduled are only known during machine start-up. This results in large-scale scheduling problems that need to be solved during start-up of the machine under a strict timing constraint on the schedule delivery time. We have incrementally dealt with several aspects of the scheduling problem in the form of four thesis contributions.

The first contribution considers directed acyclic graphs with tasks having a fixed binding on multiprocessor platforms and stringent latency requirements. It introduces a fast and scalable list scheduling approach. It uses a heuristic that makes scheduling decisions based on a new metric that finds feasible static-order schedules meeting timing requirements as quickly as possible and is shown to be scalable to very large task graphs. The computation of this metric exploits

the binding information of the application. The second contribution relaxes the fixed binding assumption while extending the model with communication costs across resources. It presents a binding algorithm that, taking these communication penalties into account, reduces the makespan of the schedule upon scheduling while ensuring that latency requirements are met.

The above contributions work under the assumption that execution times of tasks in the applications are fixed. However, tasks executing on general purpose multiprocessor platforms exhibit variations in their execution times. These variations need to be considered during scheduling to produce robust schedules that are tolerant to execution time fluctuations. The first step towards solving the robust scheduling problem is to provide a means to quantify robustness of static-order schedules. This forms the third contribution of this thesis. We define probabilistic robustness metrics that are computed using a new robustness analysis approach. Stochastic execution times of tasks are used to compute completion time distributions which are then used to compute the metrics. It overcome the difficulties involved with the max operation on distributions by presenting an approach that combines analytical and limited simulation based results. The fourth contribution is an iterative robust scheduling method that uses the above robustness analysis to produce robust multiprocessor schedules of directed acyclic graphs with a low expected number of tasks that miss deadlines. We experimentally show that this robust scheduler produces significantly more robust schedules in comparison to the scheduler from the first contribution. All of the above contributions have been validated on large industrial as well as synthetically generated test sets. The first two contributions have already been adopted into ASML's latest generation of wafer scanners.

Table of contents

Chapter 1

Introduction

With the digital revolution, modern day appliances such as smart phones, tablets, laptops, and other electronic devices have become an integral part of everyday life. They have been paramount in improving our quality of living by allowing high speed access to information and communication as well as reducing the time of our daily activities through automation. Current technological advances are slowly paving the way to the revolution of the Internet of Things [12] where more and more such objects are interconnected with each other further increasing the speed and efficiency of our day to day activities. This has been mainly possible due to the rapid advances in the semiconductor industry that has drastically reduced the amount of time needed for data processing and communication [56]. Right from the advent of the semiconductor technology, the number of transistors in an integrated circuit (IC) or chip has been growing exponentially over time. Gordon Moore had correctly predicted that the number of transistors in a chip would approximately double every two years [57]. More transistors per chip directly implies that more computing power is available to the electronic devices leading to faster growth. A modern IC can have billions of transistors on it.

Technological developments in the chip manufacturing process have made transistors smaller in size by packing more and more electronic circuitry within a small area at extremely high precision. The chip manufacturing process is a combination of chemical processing and photo-lithographic steps in which the electronic circuits are created on wafers made of a semiconductor material, mostly silicon. Chemical processing includes polishing thin slices of silicon with a photosensitive material to form silicon wafers composed of multiple chips. These silicon wafers are exposed using a lithography process that projects the circuit pattern which is placed on a quartz plate called a reticle (or photomask). This is done by passing

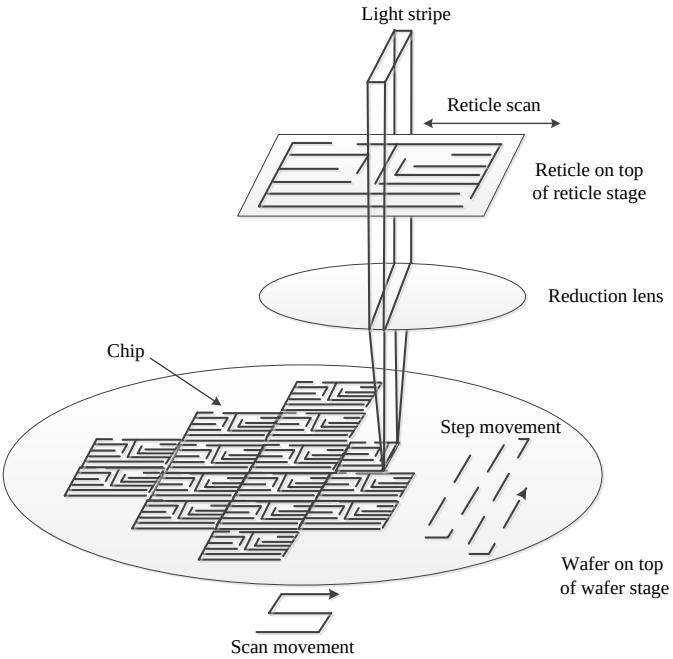


Figure 1.1: Exposure of a wafer in the step and scan fashion in the lithography process [66].

light through the reticle onto a reduction lens which reduces and projects the image formed onto the chip as shown in Figure 1.1. The exposed wafers are developed, etched and post-processed in several chemical processing steps including removal of the photosensitive layer. The whole process of exposure and wafer processing is repeated multiple times for one wafer to selectively grow, modify, and etch out every layer of the chip. A modern chip can have as many as 40 layers. Finally, the wafer is separated into individual chips that, after packaging, are used in electronic equipment.

1.1 Mechatronic control domain

The lithography process is the main step in chip manufacturing that is responsible for determining how much circuitry can be packed onto a chip, controlling the size and shape of all chip components, connections and contacts. ASML [1] is the world's leading supplier of machines, called wafer scanners, that perform this lithography process for the semiconductor industry. In order to push Moore's law, new machines are produced almost every two years with rapid advancements in the underlying technology that have allowed feature size reduction from 350

nm in 1994 to less than 20 nm today. The wafer scanners of ASML are in the forefront of technological advancements in the mechatronic control domain. This domain is composed of a combination of control, mechanical, electronic and software engineering. This combination is seen in most high tech equipment such as wafer scanners, printers, health-care devices, automotive plants, aeronautical devices or robots. Although we use the servo control systems of ASML as the primary case study in this thesis, all the thesis contributions are also applicable to several of these other domains. In this section, we look into the control and mechanical aspects of the wafer scanner systems. We further elaborate on the electronic aspects pertaining to the platform technology and the software aspects related to the model based design in the next sections.

The wafer scanners carry out lithography in a step and scan fashion on the wafer as shown in Figure 1.1 [77]. The exposure is performed by a scanning movement where the reticle and wafer move in opposite directions and the image is exposed one thin stripe at a time onto the chip. A step movement then resets the reticle to the initial position and places it on top of the next chip to be exposed. The movements required for the step and scan operations are zig-zag movements that are achieved by placing the wafer and the reticle on stages that can move in many degrees of freedom with nanometer accuracy and accelerations exceeding 20G [20]. The reduction in chip size implies that the accuracy with which the multiple chip layers are deposited on the chip and on top of each other needs to be extremely high and tolerate very low error margins. This heavily depends on the nanometer precision with which the stages are aligned with each other and perform the scanning. The same applies to several other components and robots that manipulate the wafers during the process that also need to work with extreme precision. Such precise control over the wafers is achieved through high performance control. In addition to feature size reduction, another technological driver is the throughput of the wafer scanner in terms of the number of wafers produced per hour. The accelerations with which the stages move makes the exposure process faster and thus also has an impact on the machine throughput. However, the stages tend to vibrate when subjected to such high accelerations. These disturbances need to be continuously cancelled out by means of high performance and high frequency control.

The high performance control is realized by means of a large number of closed loop or servo control systems. Servo control systems consist of control applications that are mapped onto execution platforms, as depicted in Figure 1.2. Control applications read values from sensors such as the position coordinates of the stages. They then compute mathematical functions specified in control tasks to translate these readings into corrective signals. These corrective signals are then sent to the actuators that translate them into physical actions, like movements or temperature changes. The actions are then carried out by the corresponding components to reject disturbances. These control operations are repeated periodically to ensure smooth functioning of the components. Wafer scanners have hundreds of control applications, each with hundreds of control tasks, sensors and actuators

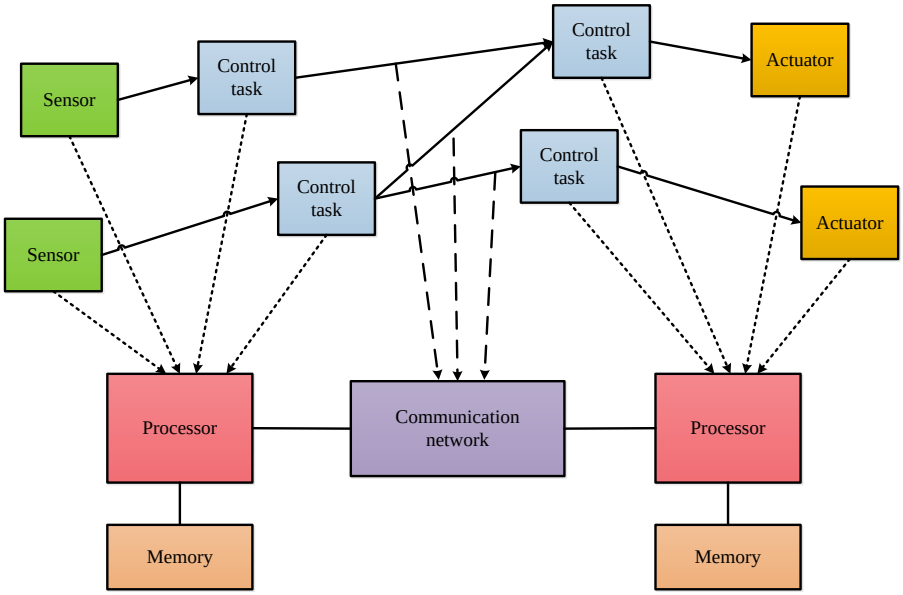


Figure 1.2: Servo control application mapped on an execution platform.

and this complexity grows with each new generation of machines. The complexity comes not only from the large number of control operations that need to be performed but also from the large number of dependencies between the control operations. The stages application, that includes the stages for the wafers and the reticle together with multiple other components, has approximately 50 connected servo controllers, 4000 process control tasks and 20,000 task dependencies. Additionally, to keep the tracking error small, these control systems need to operate at very high frequencies. This is because frequent readings and corrective actions provide more control over the system. The rate at which a control system processes sensor input samples is called the sample frequency. The stages control applications run at sample frequencies as high as 20kHz. The delay between the arrival of a sensor input sample and the generation of the actuator output sample is called the IO delay. High throughput and accuracy directly translate to higher sample frequencies and lowers IO delays. These performance requirements are assigned by control engineers in the form of constraints on sample frequency and IO delay during the design of the control systems. These constraints in turn translate into latency requirements for the control applications. A sample frequency of 20kHz translates into a latency requirement of $50\mu s$. Not meeting latency requirements thus has a direct impact on the machine throughput and the accuracy with which features can be deposited on the chips.

1.2 Platform technology

To achieve the required machine performance, control applications have to execute at high sample frequencies and have to satisfy stringent latency requirements between sensing and actuation. For this purpose a lot of computational power is needed, which is offered by multiprocessor platforms consisting of tens of homogeneous general purpose processors communicating through high bandwidth packet-switched communication networks. Partitioning of applications among the processors is often performed during design time to minimize inter-processor communication overhead over the network. For a long time there was steady increase in performance by frequency scaling of single-core general purpose processors based on Moore's law. This increase in performance has reached a limit due to the physical limitations of semiconductor devices caused by excessive heat and power dissipation resulting from leakage of currents [60]. To meet the growing demands of complex control applications, there is a shift from single-core to multi-core execution platforms. As a consequence, increase in performance of applications no longer comes for free and needs the efficient utilization of application parallelism. General purpose processors now consist of one or more processing units or cores that communicate with each other via low latency shared cache memory. Due to the relative low cost of shared memory communication, application demands can be met by exploiting their parallelism on the multi-core processors. This can be observed in the wafer scanners where only latency requirements corresponding to 10kHz and below could be met with single-core platforms. To run applications at 20kHz, multi-core platforms had to be introduced.

An important issue addressed in this thesis is the low execution-time predictability of general purpose processors. Applications running on platforms composed of these processors exhibit fluctuations in execution timings. The main reason for this unpredictability is that general purpose architectures are complex in design to make them applicable to a broad class of applications. These architectures employ complex techniques like caching, pipelining, branch predictions, and out-of-order execution to improve the performance of the processors. Caching can cause significant timing variations [63]. Execution times can range from being small during cache hits to being significantly larger when there are access delays due to cache misses. Timings variations arise also when multiple applications contend to access the shared cache memory in shared memory communication. Branch predictions aim to reduce the penalty of executing branch instructions by attempting to predict the outcome of a branch before it is known often using the history of previous outcomes. This is particularly useful in pipelined architectures where due to the pipelining of instructions the outcome of a branch statement is known much later in the pipeline. Deeper pipelines require more advanced branch prediction techniques. The timings can vary from no delays when the prediction is correct to significant delays when the predictions go wrong and the speculatively executed instructions need to be discarded to carry out the correct ones [31]. Sim-

ilarly, out-of-order execution, that attempts to reduce waiting times by executing instructions in the order based on the availability of their input data instead of the original program order, can also induce variations [42]. Hardware and software interrupts are other sources of variation wherein the execution of the current set of instructions is suspended to carry out higher priority activities requiring immediate attention.

Execution time variations severely complicate meeting the latency requirements of control applications. Alternatives to avoid variations are high performance special purpose platforms that have architectures designed to be specific to the applications running on them and have the advantage of high predictability [35]. However, the cost of making such application specific platforms for each complex type of application can be very high. General purpose platforms are popular as they are cost-effective and highly flexible. Moreover with the advent of multi-cores, they have been able to meet the demands of increasingly demanding embedded applications. The increasing popularity of general purpose platforms and their inherent unpredictability raises an important concern for the robustness of the schedules of the applications running on them to execution time variations. Robust scheduling of the tasks in the application is necessary to ensure that applications can meet latency requirements even in the presence of variations.

1.3 Model based design flow: Scheduling

Tasks of mechatronic control applications need to be bound and scheduled on multiprocessor platforms to ensure that latency requirements are met. Schedules that meet latency requirements are called *feasible schedules*. In addition to meeting latency requirements, it is also desirable to complete applications as early as possible which allows better utilization of platform resources for other applications. In the design flow used in ASML, the computation of a schedule requires information in models based on domain specific languages (DSLs) [43] that are developed by domain experts. The architecture of these models follows the Y-chart approach [46], the layers of which define the application, execution platform, binding and schedules of the application tasks on platform resources as presented in [65]. Figure 1.3 gives the model based design flow. It starts from the domain specific models of the application and the platform specification that are developed by control and hardware designers. Essential scheduling information is extracted from these models and model-to-model transformations are performed on them to construct a directed acyclic graph (DAG), consisting of control tasks and their dependencies. These DAGs form the starting point for the work of this thesis. Task deadlines are obtained from the latency requirements of the application. Although each task is aware of its processor binding, the binding among the cores of the processors is kept flexible. This is typical in ASML applications where designers tend to avoid communication costs across processors by deciding the binding on them before-hand. Application parallelism is then exploited by

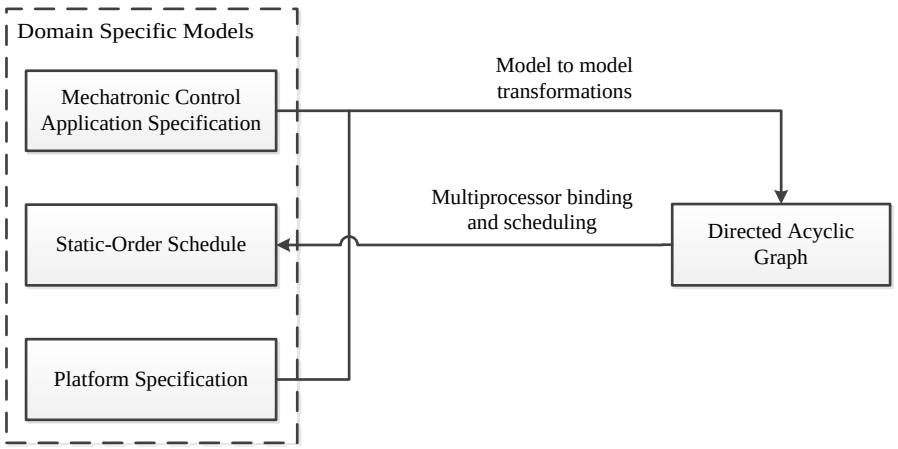


Figure 1.3: Model based design flow

utilizing the cores of the shared memory multi-core processors which tend to have relatively low communication overhead. To derive the initial binding information, the platform specification is also required as input in the transformations leading to the DAG. Depending on the platform specification the binding of the tasks can either be fully known in case of single core processors or partially known in case of multi-core processors. The binding and scheduling algorithms then transform the DAGs with their partial binding information into static-order schedules with complete binding information, i.e. orderings of tasks on the processors. These schedules are then periodically executed at their sample frequencies during run-time of the machine by a dispatcher.

The ASML wafer scanners are customized so that many of the required configuration parameters defining the characteristics of the servo control applications are available only at the start-up of the machine. As a result, the entire design flow of transforming application and platform models to DAGs and then performing scheduling needs to be done during machine start-up. The computation time of the schedules thus contributes to the startup time of machines. As the machines process as many as 300 wafers per hour, long start-up times cause a delay in the production process and can therefore be expensive to the customers. Consequently, the time to produce a schedule must be low.

Meeting latency requirements is particularly challenging due to the unpredictability of the general execution purpose platforms. Schedules produced using most-likely (referred to as 'nominal') execution times of tasks can become infeasible when tasks take longer to execute. There is a need to cope with these execution time variations and to produce schedules that are robust in nature. Ro-

bustness of a schedule is its tolerance to variations in the execution times of tasks. Scheduling and analysis of applications in classical real time approaches mostly take the worst case execution timings into account. If a static-order multiprocessor schedule of an application meets its latency requirements in the worst case, it is highly robust. However, application execution timings on general purpose platforms exhibit variations in which the nominal execution times are typically closer to the best-case execution times than to the worst-case execution times. In addition, the worst-case execution times are typically relatively large. Scheduling for the worst case then needs additional platform resources to accommodate the worst case. Moreover, it is may be very unlikely that all tasks simultaneously take an execution time close to their worst-case. Hence, the traditional worst case scheduling is pessimistic and leads to overdimensioned platforms. We thus need a stochastic scheduling mechanism that takes these execution time variations into account and produces schedules that are robust in nature.

1.4 Problem Statement and Research Challenges

The problem being dealt with in this thesis can be briefly summarized as follows:

Achieving fast, scalable, communication-aware, and robust binding and scheduling of directed acyclic graphs on multiprocessor platforms such that all latency requirements are met.

The aim is to produce static order schedules during machine start-up time. These schedules are periodically executed at their sample frequencies during run-time without changing the ordering of the tasks. The binding and scheduling problem thus focuses on a single execution of the graph and there is no notion of periodicity within the problem. To minimize the overhead of context switching during run-time, preemption of tasks is not allowed. Each task has its own deadline. These are based on the latency requirements derived from the sample frequencies. These latency requirements form the available timing budget on the processors. Deadlines assigned to control tasks can vary depending upon what are the intended operations of the tasks. For instance, tasks that contribute to sending data to actuators are assigned tighter deadlines to ensure that the sensor to actuator IO delays are kept small. Different actuators can have different delay requirements and therefore different deadlines. Remaining tasks are assigned deadlines based on the processor budget in the sampling period. The multiple challenges in the problem statement are discuss below:

1) *How can we design a scheduler that is scalable to large DAGs while still being fast?* For maximum productivity, the scheduler needs to be fast in order to produce schedules within the required start-up time window of the machines. This should be the case even for large industrial-scale input DAGs.

2) *How do we make good binding decisions in the absence of accurate information on the amount of synchronization due to communication?* In a platform consisting of multi-core processors, utilizing the parallelism in the applications and executing tasks simultaneously on parallel cores can result in significant improvement in performance. Parallel schedules produced will have lower makespans (completion time of the last task) allowing the remaining processor time to be utilized by other applications. This can be achieved by making a proper trade-off between exploiting parallelism and reducing the communication overhead that can arise when applications run on different cores. Additionally, the platform size constraints will need to be accounted for as the available resources are typically limited. Under the assumption that read-write operations are part of the task execution, shared memory communication mainly includes synchronization operations. Depending on the schedule order, many of these operations may be redundant due to the ordering being indirectly enforced by other synchronizations. Hence, the exact synchronization needed is only known after the schedule is formed and not during binding. This lack of complete knowledge makes the binding problem challenging for shared memory systems.

3) *How do we make scheduling decisions that lead to a robust schedule when the robustness of a schedule is only known after the schedule is formed?* To produce schedules that are maximally robust against execution time fluctuations, we need to design robust schedulers. This requires two steps. The first step is to precisely define robustness of tasks and schedules and to develop a means to measure this robustness. The main challenges in doing this are:

1. Finding a means to represent the execution time distributions that are skewed and bounded in nature.

2. Accounting for dependencies between task execution times.

3. Accounting for the dependencies between stochastic variables arising due to task dependencies in the graph.

4. Performing convolutions and maximization operations on distributions. There are no known practical analytical means to compute the distribution of the maximum of stochastic variables that are skewed in nature. Computing the exact max of distributions is computationally intractable.

Once we have overcome the above challenges to obtain a means to analyze robustness, the second step is to develop a robust scheduler that can use the robustness analysis to create robust schedules. The main challenge here is that the robustness of a schedule heavily depends on the robustness of its tasks, which in turn depend on the schedule order. A robust scheduler needs to predict upfront the impact of each scheduling decision on the robustness of the schedule, which is

only known when the complete schedule is formed at the end of the scheduling.

4) *How do we design a scheduler that is fast, scalable, communication aware and robust at the same time?* The final goal is to achieve a binding and scheduling mechanism that is both communication aware and robust and is still scalable to very large task graphs producing schedules in minimal time. Robustness analysis and robust scheduling are inherently computationally expensive due to the introduction of stochastic variables. Hence, the main challenge is the integration of the different aspects in combination with scalability and speed to ensure that the outcome is still applicable to the likes of the mechatronic control domain.

1.5 Contributions

In this thesis, we have broken down the problem into manageable pieces and developed solutions in an incremental manner. The thesis has four major contributions.

Contribution 1: *Fast multiprocessor scheduling with fixed task binding.* The first contribution of this thesis introduces a fast and scalable static-order scheduling approach for DAGs having tasks with deadlines and a fixed binding on multiprocessor platforms. To focus on the first aspect of the thesis problem statement, we assume that the platforms consist of single-core processors only and that the binding is thus given. Due to the fixed binding, communication overhead is assumed to be taken into account in the task execution times and is not addressed explicitly. The scheduler uses a heuristic that makes scheduling decisions based on a new due-date metric to find feasible schedules that meet timing requirements as quickly as possible and it is shown to be scalable to very large task graphs. The computation of this due-date metric exploits the binding information of the application. Experiments performed on the wafer scanner control applications allowed us to successfully verify the functioning of the scheduler in real industrial scenarios. The approach has been incorporated into all of ASML's latest generation of wafer scanners. This contribution has been published in:

[4] S. Adyanthaya, M. Geilen, T. Basten, R. Schiffelers, B. Theelen, and J. Voeten. Fast multiprocessor scheduling with fixed task binding of large scale industrial cyber physical systems. In *Euromicro Conference on Digital System Design (DSD)*, pages 979-988. IEEE, Sept 2013.

Contribution 2: *Communication aware multiprocessor binding.* In this contribution, we present a binding algorithm to compute the binding of tasks on a shared memory multiprocessor platform. The binding obtained is then fixed in the DAG and given as input to a communication aware extension of the scheduler from Contribution 1. Figure 1.4 shows this flow. It presents a three-step binding algorithm that utilizes application parallelism while taking communication

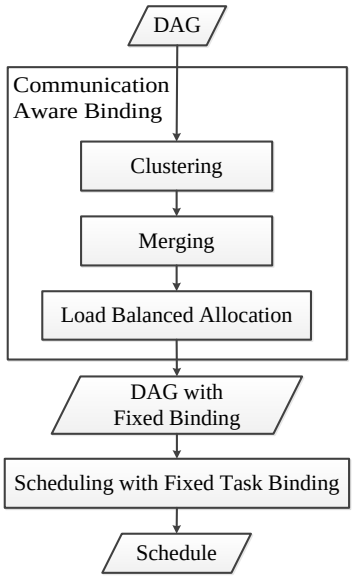


Figure 1.4: Flow of the binding results of Contribution 2 to the scheduler of Contribution 1

overhead and task deadlines into account that results in feasible schedules of the DAGs with low makespans on limited platform resources. It first clusters tasks assuming unlimited resources using a deadline-aware shared memory extension of the existing dominant sequence clustering (DSC) algorithm. The clusters thus produced are merged based on communication dependencies to make them fit on the number of available platform resources. First clustering the entire graph and then performing merging enabled us to make good binding decisions without knowing the exact amount of communication between tasks. As a final step, the clusters are allocated to the available resources by balancing the workload. The approach has been shown to outperform state of the art methods for ASML applications as well as test cases of other well-known parallel problems. This technique is about to be integrated into the ASML machines. This contribution has been accepted to be published in:

[6] S. Adyanthaya, M. Geilen, T. Basten, J. Voeten, and R. Schiffelers. Communication Aware Multiprocessor Binding for Shared Memory Systems. In Proceedings of the *11th IEEE International Symposium on Industrial Embedded Systems, SIES.* IEEE, ***Article in press***, 2016.

Contribution 3: *Robustness analysis of multiprocessor schedules.* This contribution focuses on the first step towards robust scheduling by establishing an

approach that can be used to measure the robustness of schedules of DAGs and their constituent tasks. We revert back to the assumption that the platform contains single-core processors only and that communication overhead is accounted for in task execution times. We make these assumptions to be able to focus on working towards a robust scheduler which is a complex problem in itself due to the introduction of stochastic variables. Dealing with execution time dependencies is not covered in this thesis and is a part of future work. We thus assume execution time independence. Robustness of a task is defined in terms of the probability of missing its deadline. Robustness of a schedule is then defined in terms of the (normalized) expected value of the number of tasks missing their deadlines. Lower expected value of deadline misses implies higher robustness. To compute these values, we need to propagate task execution time distributions along the dependencies between the scheduled tasks and compute task completion time distributions. This propagation requires max-plus operations to be performed on the distributions. To overcome the high complexity of doing this analytically and the drawbacks of missing rare events using simulations alone, we present a new combined analytical and limited simulations based approach. It fits a (skewed and bounded) PERT distribution on simulated histograms using analytically computed bounds. The technique has been tested for scalability and accuracy on the schedules of wafer scanner control applications that were produced using the scheduler from Contribution 1. This contribution has been published in:

[7] S. Adyanthaya, Z. Zhang, M. Geilen, J. Voeten, T. Basten, and R. Schiffelers. Robustness analysis of multiprocessor schedules. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV) 2014*, pages 9-17. IEEE, July 2014.

Contribution 4: *Iterative robust multiprocessor scheduling.* The next step, after we have been able to analyse the robustness of schedules, is to perform robust scheduling. This contribution presents a robust scheduler that uses the robustness analysis of Contribution 3 in its approach. We continue assuming single-core execution platforms. This implies that the binding is again fixed and communication overhead between processors is accounted for in task execution times. To overcome the challenge of the absence of knowledge about task and partial schedule robustness during scheduling, we present an iterative approach that iterates between scheduling and robustness analysis. This solution flow is shown in Figure 1.5. It starts from the schedule produced using the scheduler from Contribution 1 and then repeatedly performs robustness analysis followed by list scheduling with a new stochastic robustness heuristic. We quantify schedule robustness using the metric from Contribution 3 based on the expected number of tasks that miss their deadlines in the schedule. We also extend the analysis in Contribution 3 with a new metric that quantifies the task-level impact on expected deadline misses. This metric is used in an iterative highest robustness impact first heuristic to guide the stochastic list scheduler towards robust sched-

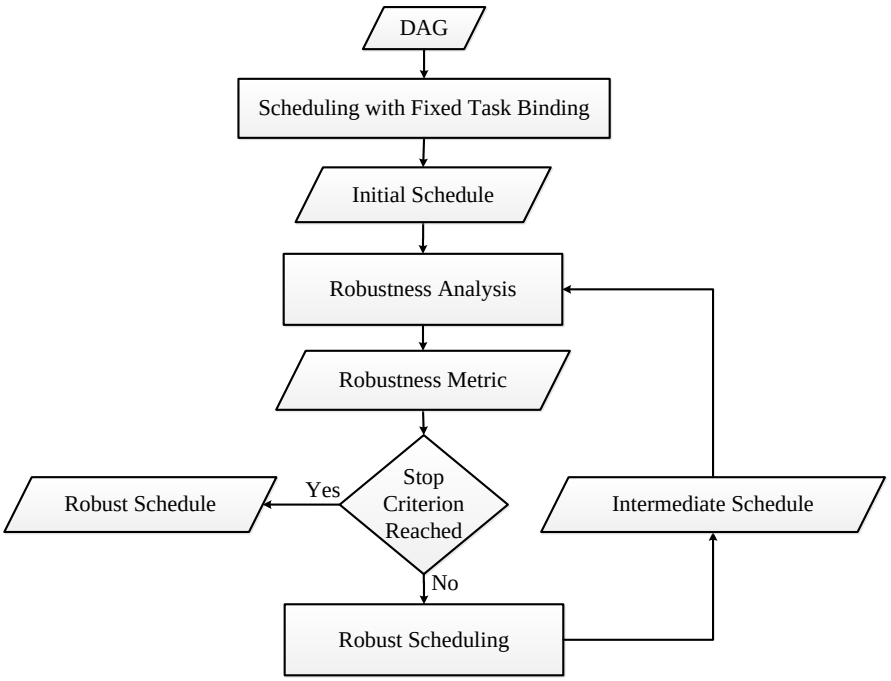


Figure 1.5: Flow of Contribution 4 starts with scheduler of Contribution 1 and iteratively uses the robustness analysis of Contribution 3

ules in the iterations. The contribution is published in:

[5] S. Adyanthaya, M. Geilen, T. Basten, J. Voeten, and R. Schiffelers. Iterative robust multiprocessor scheduling. In Proceedings of the *23rd International Conference on Real Time and Networks Systems, RTNS'15*, pages 23-32, New York, NY, USA, 2015. ACM.

1.6 Thesis outline

The remainder of this thesis is organized into five chapters. Chapter 2 presents a fast and scalable multiprocessor scheduler for DAGs with fixed task binding on multiprocessor platforms. Chapter 3 presents a binding algorithm that works in combination with the scheduler from Chapter 2 to exploit the parallelism in the application while taking communication overhead into account to produce feasible schedules with low makespans. Chapter 4 presents a new technique to

analyse the robustness of static-order multiprocessor schedules. Chapter 5 then uses this analysis in an iterative robust multiprocessor scheduling mechanism that, starting from the schedule produced using the scheduler from Chapter 2, iteratively improves the schedule robustness. Chapter 6 gives the final conclusions and directions for future work.

Chapter 2

Fast and scalable scheduling with fixed task binding

This chapter presents a scheduler that aims to rapidly obtain feasible schedules of very large applications with strict latency requirements. The contents of this chapter is based on [4]. It deals with the first thesis challenge of achieving a fast and scalable scheduler under some limiting assumptions, such as fixed binding and absence of communication cost. It uses a list scheduling heuristic based on a new due-date metric, the computation of which exploits the binding information of the application. Due-date is an upper bound on the completion time of the task which if missed makes the schedule infeasible. Computation of the optimal value for the due-date of a task is as hard as the scheduling problem itself and hence is NP-Complete. This chapter presents a simple and efficient algorithm to compute a sufficiently accurate value of the due-date in a new due-date metric that allows the scheduler to obtain feasible schedules in short time. Section 2.1 gives a motivational example for the scheduling problem. Section 2.2 presents related work in the domain of multiprocessor scheduling. Section 2.3 introduces preliminary definitions of the required basic scheduling concepts and terminology that are used in the chapter. Most of these concept are reused throughout the thesis by making slight modifications based on the context of the corresponding chapters. This section also poses the main problem statement. The complexity of the scheduling problem is analysed in Section 2.4. Section 2.5 introduces the scheduling algorithm. Experimental analysis is given in Section 2.6 and the

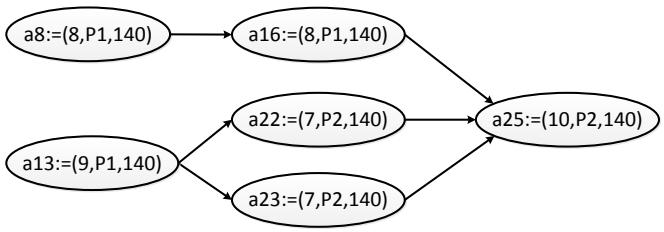


Figure 2.1: Fragment of motivational example application

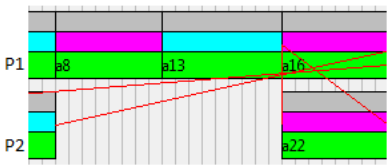


Figure 2.2: Snapshot of schedule using CALAP

summary in Section 2.7.

2.1 Motivational example

As a motivational example consider a fragment of an example application shown in Figure 2.1. A task a is defined by an execution time e, a resource r and a deadline d. For instance, task a_8 has an execution time of 8, is bound to resource P_1 and has a deadline 140. A dependency (arrow) between two tasks implies that the source task needs to be completed before the target task can begin. A task is enabled when all its predecessors have completed their execution. In Figure 2.1, both tasks a_8 and a_{13} are assumed enabled at the same time. We make a choice of which one of the two must be scheduled sooner on P_1 based on the classical as-late-as-possible ($CALAP$) completion times of the two. $CALAP$ of any task a is the minimum of the values ($CALAP_s - e_s$) of each successor s and its own deadline d_a. Although the deadline of a_{25} is 140, its $CALAP$ value is 119. It is computed by propagating backwards from the leaf nodes of the complete graph, most nodes of which have been omitted from Figure 2.1 for the sake of simplicity. In turn, a_{16}, a_{22} and a_{23}, all having a_{25} as their only successor, get a $CALAP$ value of (min(119-10, 140)) = 109. Thereafter, a_8 gets a $CALAP$ of 101 and a_{13} gets a $CALAP$ of 102. Based on $CALAP$ values, a_8 is given a preference over a_{13} and is scheduled earlier. Task a_{22} can be scheduled only after a_{13}. This results in a large gap in the schedule of P_2 shown in the snapshot of the Gantt chart in Figure 2.2, produced using visualization tools of ASML. Consequently, a later task in the schedule misses its deadline. While computing the $CALAP$ times we

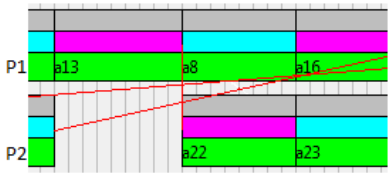


Figure 2.3: Snapshot of schedule using Due-date

do not take into account the extra information of the task bindings, thus ignoring the fact that a_{13} has higher workload following it on its resource as compared to a_8. *CALAP* computation is unaware that although a_{22} and a_{23} are parallel tasks in the graph, their binding enforces them to be scheduled sequentially on resource P_2. This binding information can be taken into account in the computation of a new bound on the task completion time referred to as due-date and denoted by dd. The due-date of a task is the minimum of $(dd_s - e_s)$ among all its successors (s) and its own deadline d_a as well as the term $(dd(S) - e(S))$ for the set S of successors of the task bound to the same resource. The term $dd(S)$ denotes the maximum among the due-dates of the tasks in S which is the due-date of the set. The term $e(S)$ denotes the sum of their execution times. This results in a due-date of (min(109-(7+7), 140)) = 95 for a_{13}. If we select based on dd instead of *CALAP*, a_{13} gets preference over a_8 resulting in a schedule with a smaller gap and no deadline misses also having a smaller makespan. A fragment of this schedule is shown in the Gantt chart in Figure 2.3.

2.2 Related work

There is a vast amount of literature available in the scheduling domain. In this section, we will first begin with a brief classification and positioning of the scheduling problem, following by bibliography and related work and lastly comparisons with closely related work. A scheduling problem is defined in terms of the characteristics of a system and the tasks that occur in it. Tasks can be classified as periodic, aperiodic or sporadic. Task priorities may be known and they may be either fixed or dynamic. Tasks can be independent or precedence-constrained. Precedence-constrained graphs without cycles are DAGs. If tasks have deadlines, then they can be soft real time or hard real time systems based on how strict the deadlines are to the real application. Optimization criteria, such as minimizing makespan, deadline misses, communication latency or optimizing throughput, are also a means of classifying scheduling problems. Based on platform characteristics, scheduling can be classified into uniprocessor and multiprocessor scheduling. Scheduling problems may be partitioned or global based on whether or not the platform binding of the tasks is decided beforehand. Furthermore, scheduling can be classified into either static-order or dynamic depending on whether or not the

entire task set is known prior to scheduling. Scheduling algorithms can be either preemptive or non-preemptive.

The problem being dealt with in this chapter is non-preemptive static-order scheduling of DAGs with task deadlines on a multiprocessor platform when the task binding is known. In literature this is often referred to as partitioned scheduling in which sets of tasks are assigned to separate processors beforehand. There has been significant research done in multiprocessor scheduling for industrial systems presented in the survey paper [30]. It discusses several partitioned and global scheduling approaches but mainly considers independent tasks. In [19], the scalability of scheduling large applications consisting of independent sporadic tasks on multi-core platforms has been addressed. It has been concluded that partitioned Earliest Deadline First (EDF) is well suited for multiple application types. In this chapter, we introduce a partitioned list scheduling approach for non-preemptive DAGs that outperforms a list scheduler which uses the earliest deadline first heuristic in the experiments. A clustered scheduling algorithm to compute binding and then scheduling of tasks for soft real-time distributed task systems with precedence constraints is presented in [54]. Another paper [51] on partitioned scheduling for multi-cores allows task pre-emption that is not allowed in our problem.

The problem of scheduling parallel applications described by DAGs on heterogenous grid computing systems is presented in [34]. DAG scheduling with arbitrary length tasks without fixed binding is known to be NP complete [59]. In this chapter, we show that the addition of the information on fixed binding does not reduce the complexity of the problem, which is still NP-Complete. Significant work has been done on DAG scheduling as presented in [50], which presents a comparison between various DAG scheduling algorithms and their complexities. List scheduling [41] is very efficient (assuming its ranking function is also efficient) and the most commonly used approach in DAG scheduling, wherein tasks are chosen from a list based on a certain priority rank and scheduled in that order. There are several heuristics that can be used to efficiently select tasks with list scheduling for better results. Some such heuristic algorithms are the Dominant Sequence Clustering (DSC) [87], Dynamic Level Scheduling (DLS) [69], Modified Critical Path (MCP), Highest Level First (HLF), Longest Path First (LP) and Longest Processing Time First(LPT) [3, 48, 84]. All these heuristics also compute binding of tasks to processing elements. The efficiency of EDF for scheduling sporadic DAG models on multiprocessors platforms with unknown binding is studied in [14]. An algorithm called *FAST* that tries to improve the quality in terms of schedule length of existing scheduling algorithms is presented in [85]. In this chapter, we use list scheduling with a new scheduling heuristic that exploits the task binding information known beforehand to make better scheduling decisions.

Work that is close to ours is presented in [80] wherein deadlines are used for the selection heuristic and a deadline modification algorithm is presented. However, this deadline modification algorithm does not take task bindings, if known, into consideration. In this chapter, it is shown that this additional information can

be used to compute tighter deadlines for tasks. In addition to this, [80] considers unit length tasks with unit communication delays. In our work we have tasks of arbitrary lengths and since the binding is known communication delays are taken into account implicitly in the task execution. In [75], classical as-late-as-possible (CALAP) time was presented which comes close to the modified deadlines of this chapter but it also does not exploit the binding information. Comparisons are made later in the chapter.

2.3 Problem definition

2.3.1 Preliminaries

We denote the sets of real numbers with $\mathbb{R}$, non-negative real numbers with $\mathbb{R}^{\geq 0}$ and integers with $\mathbb{Z}$. For a set X, we use X^* to represent the set of finite lists with elements from X.

An *application* is a DAG, $G = (T, D)$ with a finite set T of tasks and the set of dependencies between tasks $D \subseteq T^2$. A *multiprocessor platform* is a set of processors called *resources* denoted by R.

A *task* $a \in T$ is defined by a tuple $a = (e_a, r_a, d_a) \in \mathbb{R}^{\geq 0} \times R \times \mathbb{R}^{\geq 0}$, where e_a denotes the execution time, r_a denotes the resource that it is bound to and d_a denotes the deadline of a. For a set A of tasks, execution time $e(A)$ of the set is the sum of the execution times of the tasks and its deadline $d(A)$ is the largest deadline in the set.

A *dependency* $(a, b) \in D$ denotes that task b is allowed to start its execution only after the completion of task a. There can be data dependencies, control dependencies and sequence dependencies (dependencies added merely to enforce a certain sequence) in a graph.

A *(static-order) schedule* S is a mapping $S : R \rightarrow T^*$ that maps a resource to an ordered list of tasks in which all data dependencies are respected. No task begins its execution before the completion of any task having a direct or indirect dependency to it. S is a schedule for application $G = (T, D)$ iff (i) every task in T appears once in the ordered list of exactly one of the resources in S, (ii) S respects the task bindings and, (iii) dependencies. At runtime, tasks execute in the order given by the schedule as soon as their respective resource is available and their dependencies are satisfied with no pre-emptions. We let s_a denote the start time and c_a denote the completion time of task a. The start time of the first task scheduled on a resource is 0. Based on this, the values of s_a and c_a can be computed for each task a given S and G. The completion times are obtained by adding task execution times to task start times implying that $c_a = s_a + e_a$. A ***feasible schedule*** is a schedule in which all the tasks meet their respective deadlines which means that for all $a \in T, c_a \leq d_a$. During the process of scheduling, we refer to a *partial schedule* S^{prt} which is an intermediate incomplete schedule which does not necessarily contain all the tasks from the application. Note that during

run-time, tasks are scheduled in the same order as in the static-order schedule without pre-emptions. As such, even though the binding is fixed, dependencies between tasks on different processors make the multiprocessor scheduling problem more challenging than uni-processor scheduling.

The completion time of the last completing task across all resources gives the *makespan* m of the schedule defined by $m = \max_{a \in T} c_a$.

2.3.2 Problem statement

This subsection gives the statement of the exact problem being dealt with in this chapter.

Problem Statement 1. *Given an application $G = (T, D)$ with tasks bound to a set R of resources, does there exist a feasible schedule? We call this decision problem SP_{FM}, where FM stands for feasible multiprocessor scheduling.*

The execution times of tasks are fixed values extracted from measurements on the machine. Since the task bindings are known a priori, communication time is considered implicitly in the execution times of existing tasks in the task graphs and hence need not be considered separately during scheduling.

2.4 Complexity analysis

It is well known that Multiprocessor scheduling of DAGs under latency constraints with unknown binding is NP-complete [59]. This raises the question whether the additional binding information allows a more efficient solution. In [79], it is shown that determining the existence of schedules with a makespan of at most four for a collection of fork graphs with unit execution tasks, with known binding on m processors and without communication delays, is NP Complete. A fork graph is an outtree of height one that consist of a task as its source and the children of the source as its sinks. The DAGs in SP_{FM} are a generalization of a collection of fork graphs. Also, our DAGs consist of tasks with arbitrary execution times as opposed to unit execution tasks. The required makespan of at most four can be trivially converted to a task deadline of four for all tasks. All this combined with the fact that communication delays are implicit within the tasks in SP_{FM}, imply that SP_{FM} is a generalization of the problem considered in [79]. Hence, SP_{FM} is NP-hard.

The NP-Completeness of SP_{FM} can be proven by showing that it belongs to the class NP. A decision problem belongs to class NP if the solution to the problem is verifiable in polynomial time. Consider an instance of SP_{FM} and a schedule S, we need to verify if S is a feasible solution. Given the execution times of tasks, the data dependencies and the ordering of tasks on their respective resources, the completion time of tasks can be computed using a function that is linear in the number of data dependencies in the task graph. Since a graph with m tasks can

have $(m \times m)$ data dependencies, this function is quadratic in the number of tasks. This means that the completion times can be computed in polynomial time. The check for feasibility is efficient, since it involves a simple check to ensure that the completion time does not exceed the deadline for each task. This shows that given a schedule, it can be verified in polynomial time. Thus, SP_{FM} belongs to class NP and is NP-complete leading to Theorem 1.

Theorem 1. *SP_{FM} is NP-complete.*

2.5 Proposed scheduling algorithm

In this section we describe our scheduling algorithm and its heuristic to arrive at a feasible solution as quickly as possible.

2.5.1 List scheduling and static-order schedules

We perform scheduling on a per task basis using list scheduling with a new heuristic. The scheduler keeps track of the set of enabled tasks and a partial schedule. Each time, a task is chosen from a set of enabled tasks using the heuristic and is consequently scheduled on its resource. Its start time s_a is computed as the maximum of the completion times of its predecessor tasks and the earliest time that the task can be scheduled on the resource. This may lead to gaps in the schedule when a task cannot be scheduled immediately after the last scheduled task on a resource due to task dependencies. The scheduler also tries to fit tasks into existing gaps in the partial schedule. So, the earliest time on the resource also considers the earliest time among all the gaps in the partial schedule that are large enough to hold the task. Once the task is scheduled, the partial schedule is updated to include information of this scheduled task. When a task completely fits into a gap, the gap is removed from the updated schedule. If the gap is bigger than the size of the task then the task is scheduled in the gap and the gap is updated to a smaller sized gap in the schedule. Using the updated schedule, the new set of enabled tasks is computed from the data dependencies in the task graph and the process is repeated until all tasks are scheduled or the schedule is found to be infeasible and the scheduler returns with a message indicating this. This process is explained later in Section 2.5.3.

In order to improve the odds of arriving at a feasible schedule without backtracking, there is a need to smartly select which task is to be scheduled from the set of enabled tasks. The heuristic we use for this purpose is based on a derived property of a task called *due-date*, introduced next.

2.5.2 Due-dates

A due-date $dd \in \mathbb{R}$ of a task is an upper bound on the completion time of the task which, if missed, renders the scheduling problem infeasible. If dd is a due-date of

a task $a \in T$, then any $dd' > dd$ is therefore also a due-date. Also, dd is a due-date of a set A of tasks, if dd is a due-date for all $a \in A$. If specific due-dates (dd_a) are known for all $a \in A$, then the due-date of A is $dd(A) = \max_{a \in A} dd_a$. Due-dates are used to determine as soon as possible during scheduling whether a particular branch in the search space is infeasible or not hence allowing us to prune the search space by omitting the infeasible branches. They are also used in the heuristic to choose a task from the set of enabled tasks and steer the scheduling process. A task with a smaller due-date is more urgent, which leads to the scheduling heuristic called *Earliest Due-Date First* wherein a task with a smaller due-date is chosen for scheduling sooner than a task with a larger due-date. This is the same as the Earliest Due-Date heuristic (EDD) introduced by Jackson in 1955. The challenge is to find the tightest (smallest) possible due-date for a task since this enables us to detect sooner during scheduling whether a particular branch is infeasible or not. It also enables us to better determine which is the most urgent task in the set of enabled tasks and steer the scheduler in the right direction. However, finding the tightest possible due-date is as hard as solving the scheduling problem. The corresponding decision problem is NP-Complete as explained in Theorem 2.

Theorem 2. *Given a task 'a' and an arbitrary number k, deciding whether its tightest due-date dd_a is greater than or equal to k is NP-Complete.*

Proof. This theorem is proved in two steps. The first step shows that the decision problem is NP-Hard, followed by a proof of its NP-Completeness. We prove the NP-Hardness by showing a reduction from SP_{FM} which is already proven to be in NP-Complete in Section 2.4. We show that any instance of SP_{FM} can be reduced to an instance of this problem where $k = 0$. To illustrate this, consider any instance of SP_{FM} with a task graph T. We make a the initial task to T such that a has outgoing dependencies to all the source tasks in T. If there does not exist a feasible schedule to T, it implies that $dd_a < 0$. Alternatively, if there exists a feasible schedule to T, $dd_a \geq 0$. Hence, solving SP_{FM} gives the solution to an instance of this problem where $k = 0$, implying that it is at-least as hard as SP_{FM}. Hence, it is NP-Hard.

Given the tightest due-date of a and an arbitrary k, it is trivially known whether $dd_a \geq k$, implying that the solution to the problem is verifiable in polynomial time. Hence, the problem belongs to the class NP and is NP-Complete. □

Our purpose is to compute due-dates of tasks prior to scheduling and use the earliest due-date first heuristic to perform scheduling efficiently in such a manner that feasible schedules can be arrived at without the need for backtracking, hence saving time. So to simplify the due-date computation we present a means to extract a due-date from the immediate successors of a task in the task graph, since this information is easily known and does not require the exact schedule information. Computing due-dates in a particular order by traversing the task graph backwards ensures that even the future successors of a task are accounted for in its due-date computation. Along with looking at the successors, we also

look at the resources that the successors are bound to. We use this information to compute tighter due-dates as explained later in this section. The due-date so computed is the due-date we use in the earliest due-date first heuristic in this chapter.

As mentioned earlier, computation of due-dates of tasks is done starting with the sink nodes, i.e. nodes with no outgoing dependencies, in the task graph and traversing backwards in topological order of dependencies. The due-date of a task depends on its own deadline and the due-dates of its immediate successors. In the absence of any successors, the due-date of a task is simply its deadline. As mentioned in Section 2.2, a deadline modification approach is presented in [80] that does not take task bindings into account. In [75], classical ALAP times are computed by taking the minimum of the $dd_a - e_a$ for all successor tasks. However, using the binding information, successors of a task can be partitioned into groups deployed on each resource. We exploit the fact that all successor tasks on one resource must be completed before the largest of their due-dates. This is because tasks with a due-date smaller than the largest must be completed before their respective due-dates, and the tasks with the largest due-date must be completed before that largest due-date. The due date computation of a task first computes the due-date per resource, which is the due-date of the task considering only the successors bound to that particular resource, and then takes the minimum of the due-dates, so computed, over all resources. These two steps are explained below:

Due-date per resource: When a set of successors of a task is bound to the same resource, all these successors must be completed sequentially before their largest due-date. For example, in Figure 2.4, task a_1 has 3 successors. These tasks have no successors implying that their deadlines are their due-dates. Task a_2 has to be completed before time 3, task a_3 before time 4 and task a_4 before time 5. This means that all three successors must be completed before their largest due-date, 5. Task a_1 has to be scheduled before all of them and they are executed sequentially on the resource. Subtracting the sum of the execution time of all successor tasks from the largest due-date gives a due-date for a_1. Hence, 5-(1+1+2) = 1 is a due-date. By doing this, we are exploiting the additional binding information of the successors that is known to us. We also get the due-dates from looking at the $dd_a - e_a$ of the single successors: 5-2 = 3, 4-1 = 3 and 3-1 = 2. We use the minimum among all these due-dates as the due-date of the task which might be an equal or tighter value than just taking the minimum of the $dd_a - e_a$ of the individual successors. This gives us a due-date of min(1, 3, 3, 2) = 1 in Figure 2.4. Note that any subset of successors can be used to compute a due-date.

An even tighter due-date can be achieved by calculating the due-dates in this manner for all possible subsets of the successor tasks. Doing this ensures that adding an extra dependency from a task to another task in the task graph does not cause its due-date to relax. This is not the case using just the above explained computation as illustrated using the example in Figure 2.5, where a new dependency is added from task a_1 to another task a_5 in the task graph which

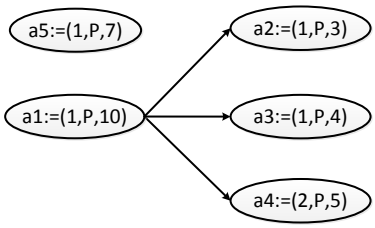


Figure 2.4: Due-date computation

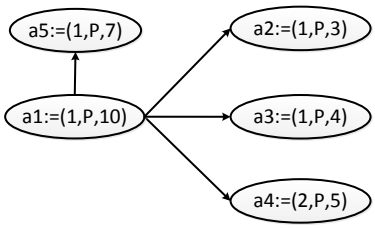


Figure 2.5: Non-monotonicity of due-date computation due to addition of a dependency

has a due-date of 7 time units and an execution time of 1 time unit. The resulting due-date of a_1 is min(7-(1+1+1+2), 7-1, 5-2, 4-1, 3-1) = 2 time units. However, we know from the earlier computation that there exists a tighter due-date which can be computed if we exclude the newly added task from the set of successors. Hence, it is necessary that this due-date computation is performed by considering all subsets of the successor tasks and taking the minimum. This computed value will be unique since we have taken into account all possible ways the successor tasks can be scheduled after the current task. In Figure 2.5, one of the subsets $\{a_2,a_3,a_4\}$ gives us the tighter due-date of 1 time unit. Computing due-dates using all successor subsets needs us to compute due-dates for all the 2^n possible subsets for n successor tasks and take the minimum among them. However, among these 2^n subsets several subsets are not necessary to obtain the due-date as elaborated in Theorem 3.

Theorem 3. *The due-date of a task with n successors computed using all 2^n successor subsets is the same as the due-date of a task computed using n subsets chosen incrementally by adding one successor at a time from the set of successors in ascending order of due-dates.*

Proof. Consider the successor with largest due-date. In the example on Figure 2.5 it is the task a_5. Keeping this task fixed in the subsets, there are 2^{n-1} possible subsets which include this task. Among these only one subset containing all successor tasks is relevant and gives the tightest due-date owing to the fact that

all successors in the subset need to be completed before the due-date of a_5. As a result we can consider this one subset instead of considering the 2^{n-1} subsets including the task a_5. This same rule applies to even the other successors of the task. For every successor we only consider one subset which contains itself and all tasks with a due-date smaller or equal to its own. We end up with exactly n subsets, one for each successor. The resultant due-date which is the minimum among the due-dates computing using these n subsets, is the same as the one obtained by taking the minimum of all subset due-dates. □

Considering subsets in the incremental manner explained above in the naive implementation for one particular task is still quadratic in the number of successors of the task since the due-date computation is repeated for each successor being added to the subset. An alternative linear approach for computation of the due-date of a task a on a particular resource r computes the same result without needing to perform the repeated computation for all the n successor subsets. It starts with an infinite due-date and sorts the successors in descending order of due-dates. Starting with the first successor in the sorted list, it assigns the $dd_a - e_a$ of this successor as the due-date of the task. Thereafter it keeps deducting the execution time of subsequent successors in the sorted list from this value as long as the result is equal to or smaller than considering the $dd_a - e_a$ of the successor being considered. In case it is larger, the value computed till then is discarded and the new due-date of the task is $dd_a - e_a$ of this successor. The execution times of subsequent successors in the list are then extracted from this value. Proposition 1 shows that this is equivalent to considering only those smaller subsets that can give a tighter due-date for the task and ignore considering all subsets in between since they do not contribute to tightening the computed due-date value in any manner.

Algorithm 1: computeDueDatePerResource()

Input : $G := (T, D), R, a, r$
Output: dd_a

```
1  dd_a := ∞;
2  for a' ∈ succ_r^ordered(a) do
3  |   if dd_a' ≤ dd_a then
4  |   |   dd_a := dd_a' − e_a';
5  |   end
6  |   else
7  |   |   dd_a := dd_a − e_a';
8  |   end
9  end
10 return min(dd_a, d_a)
```

Proposition 1. *For a task with sets A and B of successors such that $A \subset B$, the due-date computed using A is tighter only if $dd(A) < (dd(B) - e(B - A))$.*

Proof. Since $A \subset B$, B has all the successors in A plus at-least one more. The due-date computed for subset A is $dd(A) - e(A)$. The due-date of subset B is $dd(B) - e(B)$ which in turn is $dd(B) - e(B - A) - e(A)$ since $A \subset B$. As such, considering subset A will yield a tighter due-date only if $dd(A) < dd(B) - e(B - A)$. □

The minimum of the resultant value with the original deadline of the task gives its due-date. This alternative due-date computation for a particular task is linear in the number of successors to the task. Consequently, the due-date computation for the entire task graph is quadratic in the number of tasks. Let $succ_r^{ordered}(a)$ represent the list of immediate successors of a task a on its resource r, in descending order of due-dates. Algorithm 1 shows this approach.

Due-date across resources: The minimum over the due-dates computed per resource is a due-date of the task.

2.5.3 List scheduling with earliest due-date first heuristic

Algorithm 2 formalizes the scheduling process. Let all predecessors of a task a be denoted by $pred(a) := \{a' \in T \mid (a', a) \in D\}$ and all successors of a be denoted by $succ(a) := \{a' \in T \mid (a, a') \in D\}$. A gap g in the schedule, is defined by the tuple $g = (s_g, c_g) \in \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0}$, where s_g represents the start time of the gap and the c_g represents the closing time of the gap. Let G_r represent the set of gaps left on a resource $r \in R$ during scheduling. The algorithm first computes the due-dates of all tasks in the task graph in line 2. It then maintains a set of enabled tasks (ET) and another set of scheduled tasks (ST) which is initially empty. It schedules enabled tasks in a loop until all tasks in the task graph are added into the set of scheduled tasks. In line 5, the current set of enabled tasks are extracted as the ones whose predecessors have been scheduled. In line 6, the task with the earliest due-date a is chosen to be scheduled next from the set of enabled tasks. The completion time of the last completing predecessor ($lastPred_a$) of a is extracted in line 7. If there are any gaps in the schedule, the chosen gap g_{chosen} is the earliest gap that is large enough to fit a after the completion time of its last completing predecessor as given in line 8. In lines 9-11, if there is such a gap, the task is scheduled in it starting from either the beginning of the gap or somewhere in between if any of its predecessors completes later. In lines 12-15, if there are no gaps, arg min of the empty set returns $NULL$ and the task is scheduled at the end of the partial schedule on its resource either immediately after the last task scheduled on r ($last_r$) or at any later time depending on the completion times of its predecessors. In lines 16-18, if the completion time of the task exceeds its due-date, either the current or some successor task will miss its deadline and the resultant schedule will be infeasible. At this point, it could be decided to backtrack and chose tasks differently to arrive at alternative schedules.

The possibility of incorporating backtracking has not been elaborated here. If the task does not miss its due-date, it is added to the partial schedule on its resource as well as to the set of scheduled tasks in lines 20 and 21. Following this in line 22, the new set of enabled tasks are obtained by removing this task from the current set and adding all other tasks which are now enabled due to this task into the set. The whole process then repeats.

Algorithm 2: Scheduler()

Input : $G := (T, D), R$
Output: S

1 $\forall r \in R, S_r^{prt} := NULL$;
2 $computeDueDates(G, R)$;
3 $ST := \phi$;
4 **while** $|ST| \neq |T|$ **do**
5 $\quad ET := \{a \in T \mid pred(a) \subseteq ST\}$;
6 $\quad a := \arg \min_{a \in ET} dd_a$;
7 $\quad c_{lastPred_a} := \max_{a' \in pred(a)} c_{a'}$;
8 $\quad g_{chosen} := \arg \min_{g \in G(r_a)} \{s_g \mid \max(c_{lastPred_a}, s_g) + e_a \leq c_g\}$;
9 $\quad$ **if** $g_{chosen} \neq NULL$ **then**
10 $\quad\quad s_a := \max(c_{lastPred_a}, s_{g_{chosen}})$;
11 $\quad$ **end**
12 $\quad$ **else**
13 $\quad\quad c_{last_r} :=$ Completion time of last task on r;
14 $\quad\quad s_a := \max(c_{lastPred_a}, c_{last_r})$;
15 $\quad$ **end**
16 $\quad$ **if** $c_a > dd_a$ **then**
17 $\quad\quad$ **return** *'deadline miss'*;
18 $\quad$ **end**
19 $\quad$ **else**
20 $\quad\quad S_{r_a}^{prt} := append(S_{r_a}^{prt}, (e_a, r_a, d_a))$;
21 $\quad\quad ST := ST \cup \{a\}$;
22 $\quad\quad ET := \{ET \backslash \{a\}\} \cup \{a' \in succ(a) \mid pred(a') \subseteq ST\}$;
23 $\quad$ **end**
24 **end**
25 **return** S;

2.6 Experimental results

2.6.1 Industrial test cases

As mentioned in the Chapter 1, task graphs originating from the wafer scanner control domain have been scheduled using the scheduler introduced in this chapter. In these task graphs, latency requirements of the control blocks have been specified as deadlines. The binding of a block to its processor resource is given along with its execution times on the resource. Using the information in the control application, schedules are computed for each processor resource in the platform. For the purpose of our study, three separate control applications from the wafer scanner systems have been chosen as described below. The control tasks range from simple operations like addition to complex operations like filtering, clipping and matrix multiplications that are performed by larger control tasks.

1) *NXT stages.* NXT is a version of the ASML wafer scanners called TWINSCAN. These can process two wafers at a time wherein one wafer is being measured for its accurate positioning and orientation and a different wafer is being exposed with an electronic circuit. It is a collection of inter-connected applications comprising of several components that move synchronously to transfer the wafers within the system. A large number of sensors and actuators are required to perform the measurements and the movements accurately to ensure perfect wafer and reticle movement and positioning. The complete NXT stages application contains 4301 tasks and 4095 dependencies. The application graph is repeatedly executed at a sample frequency of 10kHz on a platform consisting of 11 single core processors. This frequency translates to a latency requirement of $10 \cdot 10^{-5}$ which is taken to be the available budget on the processor. The execution time of the tasks range from a minimum of $1 \cdot 10^{-8}s$ to $2.25 \cdot 10^{-6}s$. The tasks are assigned 72% of the processor budget and the remaining is kept for background processing, which gives them a deadline of $7.2 \cdot 10^{-5}s$. Apart from this, the task graphs also contain several critical tasks that send data to actuators and are assigned tighter deadlines. They are critical because the time that data is received by actuators determines the IO delay of the particular component and is crucial to the accuracy and throughput of the machine. In this case, critical actuators tasks are assigned a deadline of 30% of the processor budget giving them a deadline of $3 \cdot 10^{-5}s$. The number of incoming and outgoing dependencies (degree) of the tasks range from 0 to 22 dependencies.

Apart from this, there is also a specialized NXT stages application which assigns short deadlines to a special set of critical actuator tasks involved in a specific critical movement operation in the machine. These tasks are of higher priority and are assigned 26% of the processor budget giving them a deadline of $2.6 \cdot 10^{-5}s$.

2) *Flexray:* The lithography process to expose wafers requires light beams from a light source to be passed through the transparent and opaque regions on a quartz plate containing the images of the circuits, called a reticle, onto the wafer.

The illuminator forms a key part of this lithographic optical system. It conditions the light from the source, and causes the light beam to take on a prescribed shape, known as the pupil shape, before it goes through the reticle. This component, called the Flexray, is involved with imaging and includes 4096 mirrors working in parallel to transmit a light beam accurately onto a wafer. As such it contains an extremely large number of parallel control tasks which manipulate each mirror independently in 6 degrees of freedom to adjust the pupil shape. This is the largest model that the schedulers have been tested on, containing as many as 14,908 tasks and 26,189 dependencies. These tasks are scheduled on a platform consisting of 14 single core processors which run on a sample frequency of 238 Hz. The execution time of the tasks range from a minimum of $1 \cdot 10^{-9}s$ to as much as $8.31 \cdot 10^{-5}s$. The tasks are assigned 100% of the processor budget, since it a very large application, which gives them a deadline of $4.201 \cdot 10^{-3}s$. The degrees of the tasks range from 0 to 131 dependencies.

3) *Projection optics box (POB):* The latest NXE versions of the TWINSCAN wafer scanner systems expose wafers using extreme ultraviolet (EUV) light. The Projection Optics Box (POB) is the enclosure containing the optical components (mirrors) between the reticle and the wafer stage that holds the wafers to be exposed. Since air absorbs the EUV light, the POB is kept under vacuum to ensure that there is no loss of intensity in the light falling on the wafer. The POB models contain 1878 tasks and 1540 dependencies. These tasks are scheduled on a platform consisting of 6 single core processors which run on a sample frequency of 10kHz. The execution time of the tasks range from a minimum of $3.42 \cdot 10^{-8}s$ to $1.36 \cdot 10^{-6}s$. The tasks are assigned 70% of the processor budget, with the rest being kept for background processing, which gives them a deadline of $7 \cdot 10^{-5}s$. Critical tasks are assigned a deadline of 22% of the processor budget. The degrees of the tasks range from 0 to 13 dependencies.

Table 2.1: Measured timings for due-date computation and scheduling

Application	Due-date Computation (s)	Scheduling (s)
Stages	0.05	1.15
Sp stages	0.05	1.14
Flexray	0.18	3.64
POB	0.03	0.81

Table 2.2: Comparison between EDDF, EDF and ECF using industrial test cases

Application	EDDF	EDF	Makespan diff (s)	#Misses	ECF	Makespan diff (s)	#Misses
Stages	F	I	$9.57 \cdot 10^{-5}$	382	F	0	0
Flexray	F	I	$4.99 \cdot 10^{-4}$	563	F	0	0
POB	F	I	0	324	F	0	0
Sp stages	F	I	$9.57 \cdot 10^{-5}$	382	I	0	44

Table 2.1 gives the due-date computation and scheduling times obtained after running the scheduler on each of these control applications. To draw comparisons, we ran the above test cases with a list scheduler using the earliest due-date first (EDDF) heuristic, earliest deadline first (EDF) heuristic and the earliest CALAP first (ECF) heuristic. The first column gives the application name. The second, third and sixth columns give the outcome of whether the schedules produced were feasible (F) or infeasible (I) for EDDF, EDF and ECCF respectively. The fourth and seventh columns give the increase in makespans of EDF and ECCF respectively compared to EDDF. The fifth and eight columns give the number of deadline misses for EDF and ECCF respectively. We observed that the EDF heuristic produces infeasible schedules for all four cases. EDDF and ECF heuristics produce similar schedules in most cases. We see a difference in the specialized NXT-Motion case. This is a specialized and highly constrained application for which the list scheduler with the earliest due-date first heuristic produces a feasible schedule in the first shot as opposed to the list schedulers with the earliest CALAP first heuristic that fails to produce a feasible schedule in the first shot. The total makespans of the two schedules are the same for all applications. The results are summarized in Table 2.2.

2.6.2 Comparison of EDDF and ECF: Synthetic test cases

To evaluate the statistical significance of improvements obtained by our approach over ECF, the schedulers have been run on 1000 DAGs that are generated using the random graph generator tool of SDF3 [72]. These task graphs with 4500 tasks each are bound to 2 to 5 processors using a binding approach that binds entire branches containing subgraphs to the same resource as much as possible. This is similar to the binding approach used in ASML which binds groups of interconnected control tasks on the same resource in order to minimize communication delays across resources, as opposed to task binding where maximum parallelism is utilized. The execution times of the tasks randomly vary from $1 \cdot 10^{-2} s$ to $30s$ with an average of $2s$ in accordance with the ranges scaled up for the NXT-Motion application. The tasks are assigned deadlines which is twice the average load on the resources. Between 200-250 tasks are randomly chosen to be critical in accordance to the number of critical actuator tasks in the NXT-Motion application. These tasks are assigned deadlines equivalent to the average load on the resources. The degree of the tasks ranges from 1 to 10 with an average of 5. EDDF produced 590 feasible schedules, whereas ECF produced 517 feasible schedules. There were 336 test cases where EDDF produced feasible schedules and ECF produced infeasible schedules. On the other hand, there were 263 test cases where ECF produced feasible schedules and EDDF produced infeasible schedules. There were 254 test cases where both EDDF and ECF produced feasible schedules, and 147 cases where both produced infeasible schedules. These results are summarized in Table 2.3.

To ensure that the positive result for EDDF is not a coincidence, we performed

Table 2.3: Comparison of feasibility results of EDDF and ECF on synthetic test cases

Outcome	ECF Feasible	ECF Infeasible
EDDF Feasible	254	336
EDDF Infeasible	263	147

the McNemar test on these results and obtained a very low p-value of 0.001 which indicates that the probability that this positive outcome of EDDF is a coincidence is only 0.1%. Apart from this we also compared the makespans obtained using the two approaches. EDDF and ECF produced identical makespans in 253 of the 1000 test cases. EDDF produced a better (smaller) makespan than ECF in 403 cases and ECF produced a better makespan in 344 test cases. These results are summarized in Table 2.4. We performed the paired student t-test on the values of the makespan obtained using EDDF and ECF. A very low p-value of $7.32 \cdot 10^{-6}$ was obtained, which is also in favour of our claim that the positive outcome of EDDF is not a coincidence.

Table 2.4: Comparison of makespan results of EDDF and ECF on synthetic test cases

$\mathbf{m_{EDDF} < m_{ECF}}$	$\mathbf{m_{ECF} < m_{EDDF}}$	$\mathbf{m_{EDDF} = m_{ECF}}$
403	344	253

2.7 Summary

This chapter presents a list scheduler with the earliest due-date first heuristic to try to avoid infeasible branches in the search space during scheduling thus maximizing the odds of arriving at a feasible schedule in one shot. An efficient approach is used to compute tight task due-dates in one backward traversal of the graph by utilizing the binding information of the application on the multiprocessor platform. The key contributions of this chapter are 1) the task due-date computation technique and, 2) the overall list scheduler with earliest due-date first heuristic. The earliest due-date first heuristic has been shown to outperform the earliest CALAP first heuristic on both ASML as well as synthetic test cases. The multiprocessor scheduler has been shown to compute feasible schedules of very large task graphs within minimal time. This approach has been incorporated in all the latest versions of ASML's wafer scanners.

Chapter 3

Communication aware binding for shared memory systems

Chapter 2 presented a scheduler that assumes a fixed binding and zero communication overhead. This chapter deals with the second thesis challenge of binding by presenting a communication aware binding algorithm for shared memory systems. The contents of this chapter is based on [6]. Taking platform size constraints into account, it utilizes the application structure and communication overhead information to produce the binding of the tasks on the platform resources. In the work of this thesis, we distinguish binding and scheduling as separate concerns. The motivation behind this is to narrow down the scope of the problem on obtaining a good binding method that optimizes the makespans of the schedules while ensuring that task deadlines are met. Once, we have this binding we use the scheduler from Chapter 2 that exploits the binding to produce proper schedules. Shared memory communication is a commonly used paradigm for multi-core systems wherein multiple processors can access the same shared memory. Tasks on these processors communicate via this shared memory. As explained in Section 1.4, if we assume that read-write operations are part of the task execution, then shared memory communication mainly involves the synchronization operations only. Depending on the schedule order, many of these operations may be redundant due to the ordering being indirectly enforced by other synchronizations. For instance, a receiver task need not synchronize with a sender task if another receiver task scheduled before it on its resource has already synchronized with that particular

sender. Hence, the exact synchronization needed is only known after the schedule is formed and not during binding. This lack of complete knowledge makes binding challenging for shared memory systems.

The main contribution of this chapter is a binding approach for shared memory systems that includes three steps, namely clustering, merging and load balanced allocation. We refer to our binding algorithm as CMA ('C'lustering, 'M'erging, 'A'llocation). The first step of clustering uses a deadline-aware shared memory extension of the Dominant Sequence Clustering (DSC) algorithm [87] to produce clusters of tasks in the graph without considering platform constraints. Since the number of clusters thus formed can be higher than the number of platform resources, the next step merges clusters using the application structure while constraining the merging based on the platform size. The final step allocates clusters to resources balancing their load. After the binding, the scheduler from Chapter 2 will order the tasks on the processors. We validate CMA by drawing comparisons with the state of the art bounded dominant sequence clustering (BDSC) algorithm [44] which is an extension of DSC for limited resources. We compare them on a benchmark consisting of industrial applications of ASML wafer scanners and a large number of generated test cases of well known parallel applications. We show that our algorithm outperforms BDSC in most of the cases.

The remainder of this chapter is organised as follows. Section 3.1 starts with preliminaries, followed by a description of the problem and the solution flow. Section 3.2 explains the first clustering step. Sections 3.3 and 3.4 elaborate on the merging and resource allocation steps. Section 3.5 illustrates our experimental evaluation. Section 3.6 discusses related work. Related work has been moved to the end of this chapter, after all the concepts used in this section have been introduced, for readability purposes. Section 3.7 summarizes the chapter.

3.1 Problem definition and solution overview

3.1.1 Preliminaries

The aim of this chapter is to decide on the binding of the tasks by taking communication overhead into account. Hence, we extend the definition of a task to allow the binding of a task to be unknown and the definition of a dependency to include communication cost.

A *task* $a \in T$ is defined by a tuple $a = (e_a, r_a, d_a) \in \mathbb{R}^{\geq 0} \times (R \cup \{\perp\}) \times \mathbb{R}^{\geq 0}$, where e_a denotes the execution time, r_a denotes the resource and d_a denotes the deadline of a. The value $\perp$ is added to denote that the task is not (yet) bound to a resource. Initially r_a is $\perp$ and the binding algorithm must select and bind a to one of the resources in R.

The definition of the set of dependencies between tasks is modified to the tuple $D \subseteq (T^2 \times \mathbb{R})$ to include the communication cost. A *dependency* $(a, b, \hat{c}) \in D$ implies that it takes $\hat{c}$ time units for b to obtain the data communicated from a

only after which it can begin execution. Here, a, b are the source and destination tasks; $\hat{c}$ is the communication cost. All the other terminology is identical to the terminology introduced in Chapter 2.

3.1.2 Problem description

There are two main kinds of communication paradigms seen in practice, namely message passing systems and shared memory systems. In message passing systems tasks communicate by sending messages through a communication network and communication costs are incurred on dedicated communication resources. Our work is focused on shared memory systems where tasks communicate by reading and writing variables in a shared memory infrastructure. This happens when the tasks are bound to cores of a multi-core processor that share memory. Since shared memory read-write operations are typically non-blocking, we must ensure that the receiver task reads data from the memory only after the sender task has written into the memory. To ensure that the data flow semantics are followed, shared memory systems can make use of data synchronization mechanisms. The mechanism considered in this chapter concerns updating a status flag by the sender, indicating that it has completed writing its data. The receiver task, once enabled, waits until the status flag is set to read the data. In our work, the time taken to read and write data to the shared memory is accounted for in the execution time of the task. We make the approximation of assuming that there is no contention on the shared memory. Communication cost is thus composed of (1) the time taken by the sender task to update the status flag (U), and (2) the time taken by the receiver task to read the status update (R). The receiver task has to one by one synchronize with each sender task from which it has an incoming dependency. These communication costs are induced by communication (or synchronization) operations that are added in the schedule and executed on the same resources as the sender and receiver tasks respectively. There is no dedicated communication network and the communication costs are incurred by the computation resources. We assume communication costs to be fixed for a graph and ignore varying timings that can be caused by cache operations. Figure 3.1(a) is an example of shared memory communication, where the dotted lines show the dependencies in the graph. Task a_3 synchronizes with tasks a_1 and a_2. There is a read operation per predecessor for a_3 on its processor. Although a task needs a read operation per predecessor, it updates its status flag only once irrespective of its number of successors. Hence, the number of reads is at least equal to and can often be more than the number of updates.

The synchronization operations in schedules of shared memory systems may sometimes be redundant depending upon whether the synchronization has implicitly taken place due to synchronizations with other tasks scheduled earlier. For instance consider Figure 3.1(b) where a task a_2 is waiting to synchronize with a predecessor a_1. This synchronization is redundant if another task a_3 scheduled before a_2 on its processor has already synchronized with a_1. In this case, the

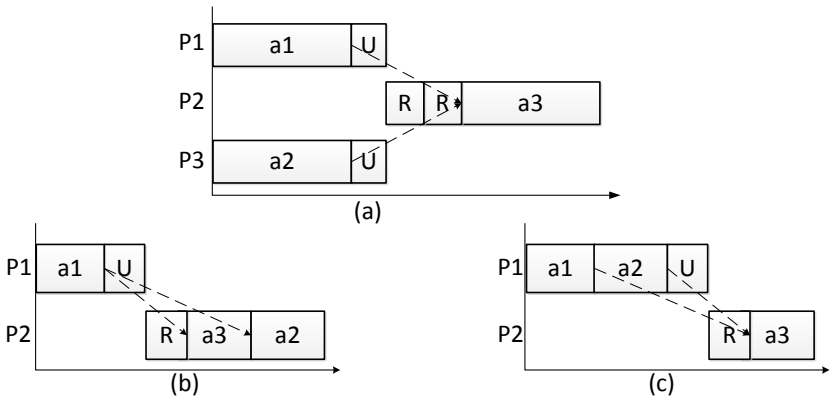


Figure 3.1: (a) Example of communication in shared memory systems, (b) No read needed for a_2 due to transitive reduction, (c) Task a_3 does not need to read the status update from a_1 and consequently a_1 does not need to update its status flag

data from a_1 is already in the memory and its flag has been updated. Then the synchronization for a_2 can be removed. This process of removing redundant communications that are implied by others is known as transitive reduction [9]. If all read operations corresponding to an update have been transitively removed, the update is also removed. This is shown in Figure 3.1(c) where the synchronization for the dependency between a_1 and a_3 is implied by that between a_2 and a_3. Task a_3 does not need to read the status update from a_1 as the status update from a_2 already implies that a_1 has finished execution. Consequently a_1 does not need to update its status flag since no task is going to read its status update. To remove such redundant synchronization, the number of communication operations needs to be optimized by performing transitive reduction on the graphs after scheduling. This is because the reduction depends on the binding and scheduling order of the tasks. For instance, in Figure 3.1(b) if a_2 is scheduled before a_3, then the read for a_3 is redundant instead of a_2. Alternatively, if a_2 is bound to a third resource P_3, then no transitive reduction is possible. Since this binding and ordering is unknown during the binding phase, it is not possible to predict the exact amount of synchronization that will appear in the final schedule. As binding algorithms typically exploit this information, this makes the binding problem challenging. The problem statement is given below.

Problem Statement 2. *Given an application and a shared memory multiprocessor platform, find a binding of the application tasks on the available platform resources that, upon scheduling, gives the lowest makespan with all task deadlines being met.*

As this problem is NP-complete, in analogy with the partitioning problem in

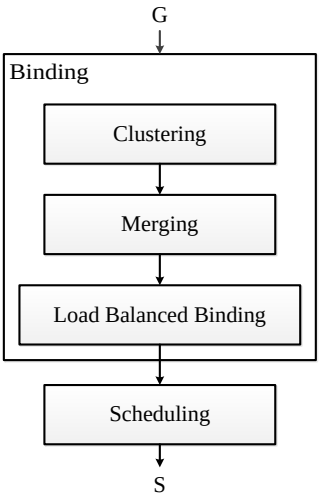


Figure 3.2: Solution Flow

graph theory [37], finding an optimal solution is not feasible. So, we present an algorithm that uses heuristics to obtain the binding of tasks.

3.1.3 Solution flow and rationale

Our solution involves three steps, namely clustering of tasks constraining them to be bound to the same resource, merging and load balanced allocation as shown in the flow chart in Figure 3.2. The benefit of having a three-step approach instead of a single step one is the use of more high level (global) structure of the graph in comparison to a single step approach like BDSC thats makes local binding decisions. After the first clustering step on unlimited resources we have the global information of all the cluster dependencies. The merging step bases its decisions on this global knowledge. BDSC, on the other hand, makes local decisions by assigning a task to one of the available clusters with the best timings for the task once the number of clusters has reached the resource limit.

The use of global information is particularly beneficial for shared memory systems because transitive reduction cannot be performed during clustering and we do not know the accurate communication costs during binding. Local decisions of BDSC made based on inaccurate timing values may not be good. We attempt to reduce the impact of this inaccuracy by first making as many clusters as possible assuming unlimited resources and then merging highly connected clusters together. The hypothesis is that there is a higher likelihood of highly connected clusters having higher communication overhead between them even after transitive reduction in comparison to less connected clusters. Hence, they should be

merged. After merging, clusters are allocated to resources by balancing the workload. The end result is a graph with known binding which is then fed into a scheduler to decide the task orderings.

3.2 Clustering

In this section we present our solution for clustering the tasks in the graph using a deadline-aware modified DSC algorithm that can deal with shared memory type of communication. We first briefly explain DSC followed by details of the extension and the algorithm itself.

3.2.1 DSC

DSC follows a list scheduling [41] pattern where at each step a new task is clustered of which all predecessors have been clustered before. We refer to such a task as an enabled task. From the enabled tasks, DSC selects a task on the longest path from source to sink in a partially clustered DAG. This path is called the dominant sequence. For a fully clustered DAG, this is the critical path. To detect which task among the enabled tasks is on the dominant sequence, DSC assigns priorities to tasks equal to the length of the longest path that passes through them. The objective of this selection is to minimize the length of the dominant sequence by avoiding communication costs along this path first. This is achieved by assigning tasks to the same clusters which in turn is expected to reduce the length of the critical path and thus the makespan. We define clusters below.

Definition 1. (Cluster) *A cluster $cl \subseteq T$ is a set of tasks that are constrained to be bound to the same resource. The size $sz(cl)$ of cl is the sum of the execution times of its constituent tasks. There exists a cluster dependency (cl, cl') if there is a dependency $(a, b, \hat{c}) \in D$ between some tasks $a \in cl$ and $b \in cl'$. We use CLD to denote the set of cluster dependencies.*

Since tasks in a cluster are assigned to the same resource, the communication costs between them is zero. The length of the longest path that passes through a task is calculated as the sum of the task's top level and bottom level, defined below.

Definition 2. (Top Level) *The top level of a task 'a' in DAG G, denoted by $tlevel(a, G) \in \mathbb{R}$, is the length of the longest path from any of the source nodes (tasks without incoming dependencies) of G to a. The length of a path is composed of the computation costs of the tasks and the communication costs of the edges along the path.*

Definition 3. (Bottom Level) *The bottom level of a task 'a' in DAG G, denoted by $blevel(a, G) \in \mathbb{R}$, is the length of the longest path from a to any of the sink nodes (tasks without outgoing dependencies) in G.*

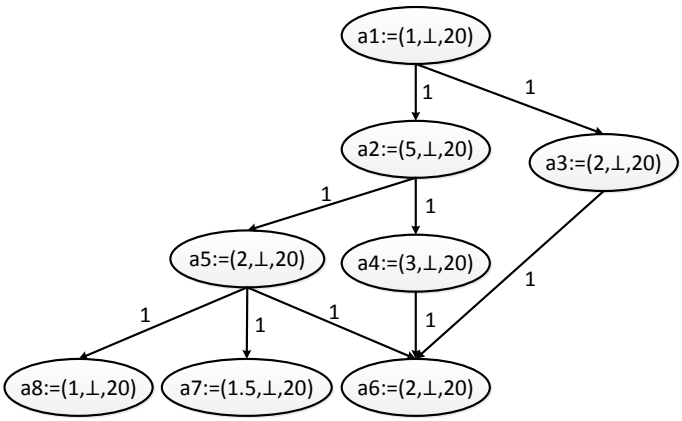


Figure 3.3: An example DAG with the binding unknown

Table 3.1: initial dsc top-levels, initial dsc bottom-levels and due-dates of tasks in the example dag

Task	***tlevel***	***blevel***	***dd***
a_1	0	14	10
a_2	2	12	15
a_3	2	5	18
a_4	8	6	18
a_5	8	5	17.75
a_6	12	2	20
a_7	11	1.5	20
a_8	11	1	20

Consider the example DAG in Figure 3.3 which needs to be bound on a platform consisting of 2 shared memory processors P_1 and P_2. The numbers shown next to the dependencies between the tasks specify the communication cost. The cost of the read and update operations is taken to be 0.5 units each giving a total communication time of 1 time unit. The initial top levels and bottom levels of tasks as computed by DSC are given in Table 3.1. The last column will be explained later. The task having the highest $tlevel + blevel$ among the enabled tasks is on the dominant sequence and is chosen to be clustered next. Priorities are assigned as $tlevel + blevel$ as this gives the length of the dominant sequence and thus has more information than using, for instance, *blevel* alone which only gives the length of the path from a current node to an exit node. In the example, the dominant sequence is a_1, a_2, a_4, a_6.

The task that is selected to be clustered is added to the cluster that gives it the

highest reduction in *tlevel* resulting from zeroing the communication cost of edges from predecessors in the cluster, if such a cluster exists. In computing the new reduced *tlevel* on the cluster, the DAG is also updated by adding a dependency from the last task in the cluster to the selected task. This is to account for the tasks that were already added to the cluster before the current selected task. These additional dependencies in the graph are temporary and are used only within the clustering phase for *tlevel* computations. If there is no such cluster that reduces the *tlevel*, the task is assigned to a new cluster and its *tlevel* remains the same. After each clustering step, the top levels of un-clustered tasks must be recomputed taking into account the changes in the *tlevel* of their clustered predecessors. The complexity of performing this re-computation per step can be reduced by incrementally computing and updating top levels of enabled tasks.

3.2.2 Deadline-aware extension to DSC

In our work, tasks in a DAG have deadlines that must be met. These deadlines must be taken into account for task priorities to avoid clustering decisions to cause deadline misses. In our extension to DSC, we compute priorities based on top levels and the due-dates from Chapter 2. The computation of due-dates utilizes the binding information of the tasks in the graph to compute tight bounds. However, before clustering, the binding of tasks is unknown. Hence, we generalize the due-date computation of Chapter 2 to one that does not know the exact binding of tasks. The due-date of a task a depends on its own deadline and the due-dates of its immediate successors. If the task has no successors, its due-date is simply its deadline. Given the due-dates of all the successors, the due-date of a task with unknown binding is computed using Equation 3.1, where $succ(a)$ refers to the set of all successors of a.

$$dd_a = \min \left\{ d_a, \min_{s \in succ(a)} (dd_s - e_s), \left(\max_{s \in succ(a)} dd_s - \frac{\sum_{s \in succ(a)} e_s}{|R|} \right) \right\} \tag{3.1}$$

This equation is composed of three terms:
1) The due-date of a task is at most its deadline d_a.
2) Each successor of a has to complete before its due-date. Hence, a must finish its execution before the minimum of $(dd_s - e_s)$ over all the successors.
3) The third term exploits the fact that all successors must complete before the maximum of their due-dates. For example in Figure 3.3, all successors of a_5 must be completed before the maximum of their due-dates equal to 20. The execution time for the total workload cannot be less than the averaged workload over the set of resources. Hence, a must be completed before the average successor workload deducted from the maximum of the successor due-dates. The averaged workload of the successors of a_5 over the set of resources $\{P_1, P_2\}$ is $4.5/2 = 2.25$. Hence, the value of this term for a_5 is $20 - 2.25 = 17.75$. The due-date of a is the minimum of these three terms. Just like in Chapter 2, due-dates of all tasks are

computed before the start of the binding process by a single backward traversal through the graph. For Figure 3.3, starting from leaf nodes a_6, a_7 and a_8 and having their deadlines as due-dates, the remaining task due-dates are computed using Equation 3.1 in the last column of Table 3.1.

Since tasks are not yet scheduled during the computation of due-dates, it is not possible to have a good estimate of the amount of communication penalty that needs to be taken into account during the computation. Hence, we compute a conservative estimate of the due-dates by not taking any communication overhead into account. Given the task due-dates and top levels, we assign priorities based on the slack sl between their due-dates and their top levels increased with their execution. The smaller the slack, the higher the priority.

$$sl_a = dd_a - (tlevel(a, G) + e_a) \tag{3.2}$$

Algorithm 3: Clustering()

Input : $G := (T, D)$
Output: CL, CLD

$CT := \emptyset$, $CL := \emptyset$, $CLD := \emptyset$;
while $CT \neq T$ **do**
  $ET := \{a \in T \mid pred(a) \subseteq CT\}$;
  $\forall a \in ET, tlevel(a, G) :=$ getTLevel$(a, \emptyset)$;
  $a := \arg\min_{a \in ET} dd_a - (tlevel(a, G) + e_a)$;
  $cl_a := \{cl \in CL \mid$getTLevel$(a, cl) < tlevel(a, G)\}$;
  if $cl_a = \emptyset$ **then**
    $cl_a :=$ createNewCluster();
    $CL := CL \cup \{cl_a\}$;
  end
  $tlevel(a, G) :=$ getTLevel(a, cl_a);
  for $a' \in pred(a)$ **do**
    if $cl_{a'} \neq cl_a$ **then**
      $CLD := CLD \cup \{(cl_{a'}, cl_a)\}$
    end
  end
  $CT := CT \cup \{a\}$
end
return CL, CLD;

The clustering algorithm is given in Algorithm 3. The due-dates of all tasks are computed before clustering using Equation 3.1. As a pre-processing step, we perform transitive reduction on the DAG thus removing dependencies that are already redundant in the graph. CL represents the set of formed clusters, ET

represents the set of enabled tasks and the set CT holds all clustered tasks. In the beginning the sets CL and CT are empty. Recall that we refer to the set of all predecessors of a task a as $pred(a)$. We compute the top levels of all tasks from the list of enabled tasks in line 4. The highest priority enabled task is chosen in line 5. Line 6 assigns it to the cluster that gives the lowest updated top level that is below its current top level. If there is no such cluster then a new cluster is created with the task in lines 7-10. Line 11 updates the *tlevel* of the task to that on its chosen cluster. Lines 12-16 add a cluster dependency for every dependency between the chosen task and its predecessors on other clusters. In line 17, the task is added to the set of clustered tasks. This process repeats until all tasks have been clustered. The algorithm returns the set CL of clusters and the set CLD of cluster dependencies. Its complexity is $O((|T| + |D|)log|T|)$, which is the same as the complexity of DSC [87].

3.2.3 Shared memory extension to DSC

In this section, we adapt the *tlevel* computation in the clustering algorithm of DSC, which was designed for message passing behaviour, to deal with shared memory systems. Communication operations are scheduled on computation resources in shared memory systems. As no transitive reduction can be performed during the top level computation, we need to make useful approximations about the synchronization costs to be considered to make better timing predictions. Firstly, we only consider read operations and ignore the update operations in the top level computations. This is because there is at most one update operation per sender as opposed to multiple possible read operations per receiver corresponding to all its predecessors. Hence, the read operations form the primary varying factor in the communication costs. Due to the lower impact of the update operations and the likelihood that the update operation will be removed later during scheduling and transitive reduction, we make the approximation of not considering them at all. Algorithm 4 gives the computation of the top level of task a for shared memory systems. If no cluster is given as input, the algorithm returns the top level of a before assigning it to a cluster. If a cluster is given as input, the algorithm returns the top level of a in the cluster. In this case, it adds a dependency from the last task in the cluster to a in lines 1-6. Line 7 computes the maximum of the sum of top level and execution times of the predecessors of a. The top level is obtained by adding the total communication cost to this value. When no cluster is specified, then the total communication cost is the sum of the communication costs from all predecessors in lines 8-10, where *commCost* is the cost of one read operation. When a cluster is specified, the total communication cost is the sum of the communication costs for only the predecessors in other clusters in lines 11-17. We take the sum of the communication costs (as opposed to maximum in DSC) because the synchronization operations are executed on the same resource as the task.

Table 3.2 shows the process of the clustering algorithm on the DAG in Fig-

Algorithm 4: getTLevel()

Input : a, cl
Output: $tlevel(a, G)$

```
1  if cl ≠ ∅ then
2  |   a_last := last task clustered in cl;
3  |   if a_last ≠ ∅ then
4  |   |   D := D ∪ {(a_last, a, 0)};
5  |   end
6  end
7  tlevel(a, G) := max_{p∈pred(a)} (tlevel(p, G) + e_p);
8  if cl = ∅ then
9  |   tlevel(a, G) := tlevel(a, G) + commCost * |pred(a)|;
10 end
11 else
12 |   for p ∈ pred(a) do
13 |   |   if cl_p ≠ cl then
14 |   |   |   tlevel(a, G) := tlevel(a, G) + commCost;
15 |   |   end
16 |   end
17 end
18 return tlevel(a, G);
```

ure 3.3. The first column gives the clustering step. The corresponding task chosen in that step, its top level before being assigned to a cluster for that step, its due-date and its slack are given in the second to fifth columns. The remaining columns give the top levels in the available clusters with the * indicating the cluster that is chosen. In the first step only task a_1 is enabled and is added to a new cluster cl_1. After a_1 is clustered, a_2 and a_3 become enabled. As a_2 has a lower slack it is chosen to be clustered on cl_1 which gives a reduction in its top level. Notice that the top level of a_2 is 1.5 as we have only considered the read part of the communication (explained earlier) which is 0.5 time units. Next, a_4 is clustered on cl_1 after which a_5 is added to a new cluster cl_2 since cl_1 increases its top level. Then, a_7 is added to cl_2 and a_8 is added to a new cluster cl_3 as cl_1 and cl_2 increase its top level. The next task a_3 has a much smaller top level than on both cl_1 and cl_2 and is added to a new cluster cl_4. Although a_3 has been enabled since step 2, it was not chosen until step 6 due to its low priority resulting from its high slack. Finally, a_6 is added to cl_1. In Section 3.3, we will attempt to merge some of these clusters to reduce them to the number of resources.

Table 3.2: clustering algorithm on the example dag

Step	Task	*tlevel*	*dd*	*sl*	Top levels on clusters			
					cl_1	cl_2	cl_3	cl_4
1	a_1	0	10	9	0*			
2	a_2	1.5	15	8.5	1*			
3	a_4	6.5	18	8.5	6*			
4	a_5	6.5	17.75	9.25	9	6.5*		
5	a_7	9	20	9.5	9.5	8.5*		
6	a_8	9	20	10	9.5	10	9*	
7	a_3	1.5	18	14.5	9	10.5	10.5	1.5*
8	a_6	10.5	20	7.5	10*	11	11.5	10

3.2.4 BDSC

BDSC is a one step approach for binding of DAGs on a limited number of resources [44]. It uses an additional heuristic that checks for and fills up the idle slots at the end of existing clusters to limit the number of new clusters created. Once the number of clusters reaches the resource limit, it assigns tasks to one of the available clusters that gives it the lowest top level. In [44], it is shown to outperform other binding methods for limited resources and is therefore our benchmark for comparisons. Table 3.3 gives the outcome of applying BDSC to the DAG from Figure 3.3. The top level computations use the full communication cost of 1 time unit here. The clustering until step 5 is the same as in our algorithm. In step 6, task a_8 chooses cl_1 since BDSC disallows new clusters to be created after the resource limit is reached. Next, a_3 and a_6 are clustered into cl_2 finally resulting in tasks a_1, a_2, a_4, a_8 being bound to one processor and a_3, a_5, a_6, a_7 to the other.

Table 3.3: bdsc on the example dag

Step	Task	*tlevel*	*blevel*	*DS*	Top level on clusters	
					cl_1	cl_2
1	a_1	0	14	14	0*	
2	a_2	2	12	14	1*	
3	a_4	8	6	14	6*	
4	a_5	8	5	13	9	7*
5	a_7	11	1.5	12.5	10	9*
6	a_8	11	1	12	10*	10.5
7	a_3	2	5	7	11	10.5*
8	a_6	12	2	14	13.5	12.5*

3.3 Merging

Algorithm 3 produces clusters assuming unlimited resources in the platform. The merging step combines highly connected clusters to bring down the number of clusters towards the number of available resources. The criterion used for merging is based on the number of cluster dependencies. Highly mutually connected clusters are merged to save the communication costs between resources after the binding.

Definition 4. (HIGHEST CONNECTED CLUSTER) *A highest connected cluster function maps a cluster cl with one of the clusters it has the highest number of cluster dependencies with. It is denoted by* $HCC : CL \rightarrow CL$. *If the cluster has no dependencies to any other cluster, it returns null.*

Algorithm 5: Merging()

Input : CL
Output: CL_{Merged}

```
1  if |CL| ≤ |R| then
2  |  return CL;
3  end
4  else
5  |  CL_Sorted := Clusters sorted in descending order of number of cluster
   |  dependencies with their HCCs;
6  |  for cl ∈ CL_Sorted do
7  |  |  if HCC(cl) = null then
8  |  |  |  return CL_Sorted
9  |  |  end
10 |  |  if sz(cl) + sz(HCC(cl)) ≤ threshold then
11 |  |  |  CL_Sorted := CL_Sorted ∪ {(cl + HCC(cl))};
12 |  |  |  Update CLD, HCC, CL_Sorted;
13 |  |  |  CL_Sorted := CL_Sorted\{cl, HCC(cl)};
14 |  |  |  Update CLD, HCC;
15 |  |  end
16 |  |  if |CL_Sorted| = |R| then
17 |  |  |  return CL_Sorted;
18 |  |  end
19 |  end
20 |  CL_Merged := CL_Sorted;
21 |  return CL_Merged;
22 end
```

The merging algorithm is given in Algorithm 5. It takes the set CL of clusters

and returns the set CL_{Merged} of merged clusters. If the number of clusters is less than or equal to the number of resources, we stop the process in lines 1-3. In line 5, the clusters are sorted in descending order of the number of cluster dependencies they have with their highest connected clusters. When a cluster has the same highest number of dependencies with more than one cluster we choose arbitrarily. Counting the number of cluster dependencies instead of the costs of these dependencies is sufficient here because the communication costs are proportional to the number of dependencies due to the costs being fixed. The algorithm then chooses the highest connected cluster from the sorted list. If this cluster has no highest connected cluster, we have only clusters left with no cluster dependencies. The process then stops and returns the clusters. Note that there can still be more clusters than the number of resources. Otherwise it merges the chosen cluster with its highest connected cluster if the sum of the two clusters does not exceed a *threshold* given by the total workload of the task graph divided by the number of resources.

$$threshold := \frac{\sum_{a \in T} e_a}{|R|} \tag{3.3}$$

This enforces that the combined size of the merged clusters does not exceed the average workload of the graph on the given resource set. The motivation for this threshold is to ensure that not too many clusters are merged together resulting in unevenly sized clusters whose workload cannot be evenly balanced on the resources thus adversely affecting the makespan. If the combined size is within the threshold, the chosen cluster is merged with its highest connected cluster while updating *CLD*, *HCC* and the sorted cluster list accordingly in lines 10-15. If the number of resulting clusters are equal to the number of resources, the clusters are returned in lines 16-18. Otherwise, the process repeats until the end of the list and returns the clusters. Since we sort the clusters in this algorithm, each time two clusters are merged the sorted list along with the relevant cluster dependencies need to be updated to include the new cluster and remove the constituent ones. Hence, the worst case complexity of this algorithm is $O(|T|(|T| + |D|))$. In the example of Figure 3.3, the average workload is $17.5/2 = 8.75$ time units. On top of the sorted cluster list is cl_1 whose size is 11 time units which is larger than the threshold. Hence it cannot be merged despite having dependencies with other clusters. Next in the list is cl_2 which has one dependency with its highest connected cluster cl_3. Their combined size is 4 which is below the threshold and they are merged into cl_{23}. Next cl_4 remains in a separate cluster as it has no dependencies with cl_{23} and it cannot be merged with cl_1 due to the threshold.

3.4 Load balanced allocation

After merging, the clusters are allocated to the available resources while balancing the workload. This step, detailed in Algorithm 6, ensures that clusters are evenly

Algorithm 6: LoadBalancedAllocation()

Input : CL_{Merged}, R
Output: CA

1 $\forall r \in R, capacity(r) := \sum_{cl \in CL_{Merged}} sz(cl)$;
2 CL_{Sorted} := Clusters sorted on decreasing size;
3 **for** $cl \in CL_{Sorted}$ **do**
4 $\quad r := \arg\max_{r \in R} capacity(r)$
5 $\quad CA := CA \cup (cl, r)$;
6 $\quad capacity(r) := capacity(r) - sz(cl)$;
7 **end**
8 **return** CA;

distributed on the cores. The cluster allocation function is defined below.

Definition 5. (Cluster Allocation) *A cluster allocation is a function which allocates a cluster in CL to a resource in R. It is denoted by* $CA : CL \rightarrow R$.

At first, all resources in R are assumed to have maximum capacity equal to the sum of the sizes of all clusters in line 1. Clusters are sorted in decreasing order of size in line 2. The largest cluster is then allocated to the resource having the highest remaining capacity in lines 3-5. The size of the cluster is then deducted from the capacity of the chosen resource in line 6 and the process repeats until all clusters are allocated. When a cluster is allocated to a resource $r \in R$, all tasks in the cluster are bound to r. This algorithm has linear complexity. The overall complexity of CMA comes from the merging step and is equal to $O(|T|(|T| + |D|))$. In the example, the allocation step assigns cl_1 to one resource and cl_{23} together with cl_4 to the other resource. This gives us the final binding where tasks a_1, a_2, a_4, a_5 are bound to one processor and tasks a_3, a_5, a_7, a_8 to the other.

3.5 Experimental Setup and Results

In this section, we evaluate CMA by comparing its binding to BDSC on industrial test cases and other well-known test cases. For the evaluation, we need the scheduler to order the tasks once they are bound to resources. We first elaborate on this scheduler setup.

3.5.1 Scheduler Setup

For scheduling, we adopt the list scheduler with the earliest due-date first heuristic from Chapter 2. By recomputing the due-dates after the binding technique

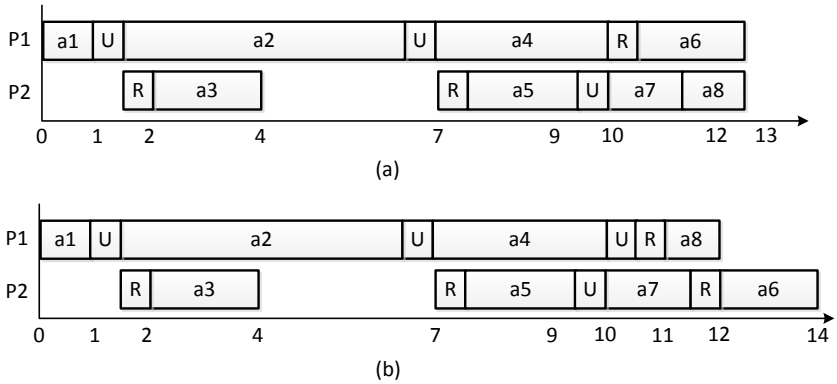


Figure 3.4: Schedules obtained from (a) CMA, (b) BDSC for the example DAG

presented in this chapter, we can exploit the task bindings so computed in the due-date computations and get tight bounds on the completion times. However, this scheduler does not take communication overhead into account to compute start times of tasks. A small modification in the computation of the start times is needed to take the communication overhead into account. We perform transitive reduction at each scheduling step based on the partial schedule to get an estimation of the number of synchronization operations that need to be taken into account in the start times using Equation 3.4.

$$s_a = \max(\max_{p \in pred(a)} c_p, \max_{a' \in T \& (r_{a'} = r_a)} c_{a'}) + tranRedCommCosts \tag{3.4}$$

Here, $\max_{p \in pred(a)} c_p$ and $\max_{a' \in T \& (r_{a'} = r_a)} c_{a'}$ give the completion time of the last completing predecessor of a and the last task scheduled on r_a respectively. The term $tranRedCommCosts$ refers to the transitively reduced synchronization costs.

For the DAG in Figure 3.3, the resultant schedule after the ordering of tasks by the scheduler and the transitive reduction of the redundant synchronization is given in Figure 3.4(a). It has a makespan of 12.5 time units. Figure 3.4(b) gives the schedule obtained from BDSC which has a makespan of 14 time units.

3.5.2 Results

We now compare CMA to BDSC on some industrial and other well known application graphs. We use the scheduler of the previous section for both methods and evaluate the binding based on the resultant makespans and deadlines being met. The features that we use to categorize the test cases are as follows:

- Communication to computation ratio (CCR): This is the ratio of the communication cost to the average computation cost (total of task execution

times divided by the number of tasks). A low CCR value implies that the application is computation-intensive. We have generated test cases for CCR values in {0.1, 0.5, 1, 2, 3,..., 10} by adapting the task execution times.

- Number of processors (#P) the application is to be bound to. We have chosen a range of 2-16 processors.
- Size of the application: We quantify application size in terms of its number of tasks. This depends on the kind of application and its attributes, explained separately below.

Industrial applications. We applied the binding approach on three large control applications of the ASML wafer scanners. The architecture of the platforms consists of homogenous general purpose single-core and octo-core processors. Tasks on the octo-core processors use the shared memory communication mechanism between cores. Communication between processors is carried out through a communication network. Due to the relatively high cost of communication through the network compared to shared memory, partial binding of the application tasks on the processors is still manually decided by the designers to keep the communication through the network minimal. The remaining binding of the tasks on the cores of the multi-core processors is to be computed using CMA. Multi-core processors in the platform architectures of ASML are typically octo-cores. One of the cores of the octo-core processors is reserved for background processing implying that tasks bound to the octo-cores can choose from the remaining seven cores. Additionally, the tasks in these applications are subdivided into operational phases based on criticality. If there exists a dependency from phase A to B, then all tasks in phase A should complete execution before the beginning of phase B. Hence, we have performed the binding of the phases separately and reported the final makespan results. Some phases do not have much parallelism to be exploited; others have very few and relatively independent tasks. These corner cases have been excluded from binding. During scheduling, the scheduler binds tasks in the excluded phases to the core with the earliest start time. Synchronization cost is fixed to $1.95 \cdot 10^{-9} s$ for the ASML platform to abstract from jitter and caching delay variations. Task execution times and synchronization costs are taken as the average of measurements.

Note that these test cases are different from those in Chapter 2 as the control applications were updated during the introduction of multi-cores and we worked with the latest versions. The dependencies are also much higher in these applications because the dependencies between phases are specified in a different manner when dealing with multi-core platforms. Applications also differ based on the version of the machines that they are part of. The applications are detailed below.

1) *NXE stages.* We took the stages application from the latest NXE 3350B version of the TWINSCAN wafer scanners of ASML that expose wafers using EUV light. It has 4,438 control tasks and 48,491 dependencies, with a degree

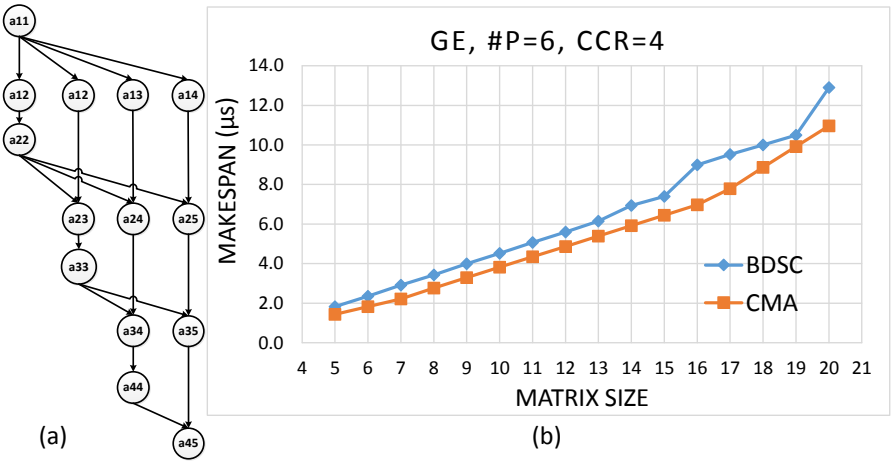


Figure 3.5: (a) Example graph for GE, (b) GE results for the given matrix sizes

(total incoming and outgoing dependencies) ranging from 1 to 240. The application graph is repeatedly executed at a sample frequency of 20kHz. This frequency translates to a latency requirement of $5 \cdot 10^{-5}s$ which is taken to be the available budget on the processor. The platform consists of eight octo-cores. Task deadlines, decided based on their criticality by domain experts in ASML, range from 50% to 55% of the processor budget for critical tasks and 100% of the processor budget for non-critical ones. Task execution times range from $5 \cdot 10^{-10}s$ to $2.2 \cdot 10^{-5}s$ with an average of $1.1 \cdot 10^{-7}s$. The CCR values range from 0.009 to 383 with an average of 11. CMA produces a makespan of $3.61 \cdot 10^{-5}s$. This is 3% lower than the $3.73 \cdot 10^{-5}s$ obtained from BDSC. The completion times of the last completing tasks per multi-core processor from CMA are 0-9% lower than BDSC.

2) *FlexRay.* We chose the latest Flexray application from the NXT version of TWINSCAN. It consists of 13,950 tasks and 663,141 dependencies, with the degree of the tasks ranging from 1 to 1,233, executed at a sample frequency of 238Hz. The platform consists of four octo-core processors. Task deadlines range from 50%-55% of the processor budget for critical tasks and are equal to 100% for non-critical tasks. Task execution times range from $1.8 \cdot 10^{-8}s$ to $1.5 \cdot 10^{-6}s$ with an average of $1.9 \cdot 10^{-7}s$. The CCR values range from 0.1 to 11 with 1.4 being the average. CMA produces a makespan of $3.41 \cdot 10^{-3}s$ which is 0.6% lower than $3.43 \cdot 10^{-3}s$ from BDSC. The completion times of the last tasks per multi-core processor are lower by 0.3% to 14.7%.

3) *POB.* We chose the POB application from the TWINSCAN NXE 3350B version of the wafer scanners. It consists of 2,769 tasks and 35,617 dependencies, with a degree in the range 1-158. The application is composed of two independent parts. One part is executed at a sample frequency of 20kHz and bound to three

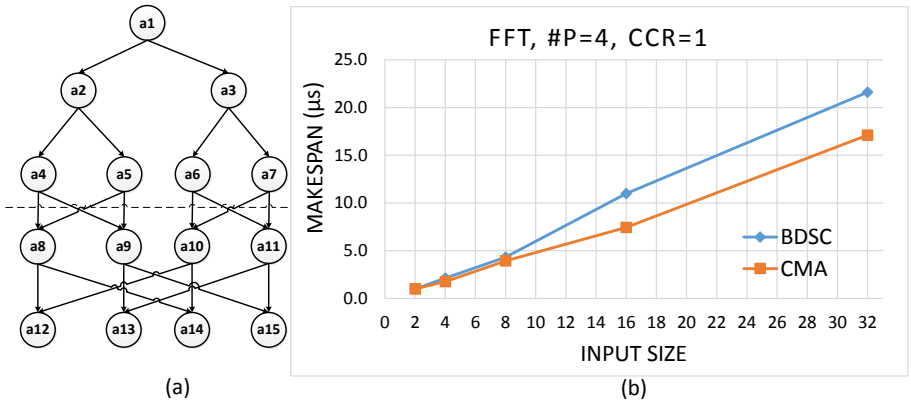


Figure 3.6: (a) Example graph for FFT, (b) FFT results for the given input sizes

octo-core processors. The other is executed at 10kHz and bound to two single-core processors. Deadlines of tasks range from 50% to 55% of their respective processor budget for critical tasks and 100% of the budget for non-critical tasks. Task execution times range from $2.5 \cdot 10^{-10}s$ to $1.8 \cdot 10^{-6}s$ with an average of $1 \cdot 10^{-7}s$. CCR values range from 0.1 to 774 with an average of 20. The makespan of $2.16 \cdot 10^{-5}s$ from CMA is 6% lower compared to the $2.31 \cdot 10^{-5}s$ from BDSC. The completion times of the last tasks on the multi-core processors are lower by 5-9%.

In all of the above applications task deadlines were met by both approaches. However, CMA produces a lower makespan in each case. As we know that BDSC only focuses on makespans and does not account for task deadlines, we need another experiment to make a fairer comparison focusing on makespan results alone. We henceforth abstract from deadlines by assigning equal and very large deadlines to all tasks.

Application graphs of well-known problems. We generated applications graphs of varying sizes bound to varying sized platforms for four kinds of well-known problems. For each case, we chose to show a result plot for one set of input values that is representative of the typical outcome observed.

1) *Gaussian Elimination (GE).* Gaussian elimination is an algorithm that solves a matrix of linear equations to return the values of the unknown variables [27]. For a matrix of size m, the corresponding task graph has $\frac{m^2+m-2}{2}$ tasks. Figure 3.5(a) is the task graph for a matrix size of 5. The critical path is $a_{11}, a_{12}, a_{22}, a_{23}, a_{33}, a_{34}, a_{44}, a_{45}$ having the most tasks. We generated a wide range of test cases for matrix sizes from 5 to 20. The outcome for a platform of 6 processors and a CCR of 4 over all matrix sizes is in Figure 3.5(b). We see that CMA always produces a lower makespan than BDSC.

2) *Fast Fourier Transform (FFT).* Fast fourier transform [26] is an algorithm

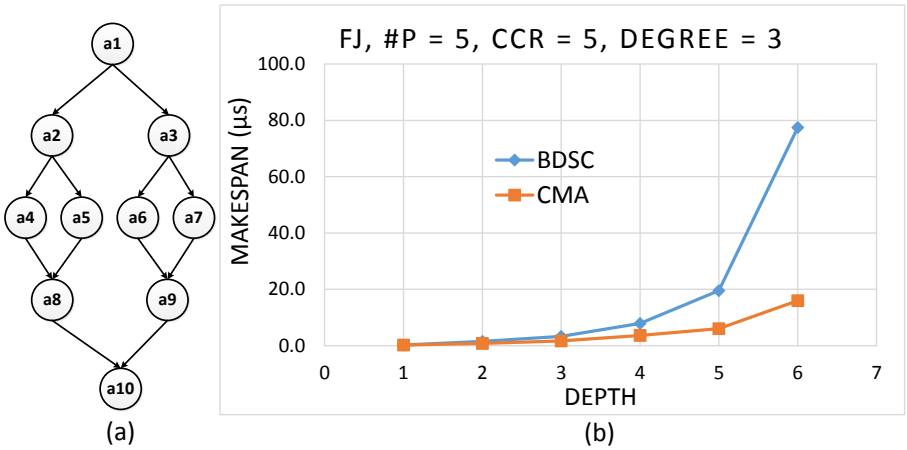


Figure 3.7: (a) Example graph for FJ, (b) FJ results for the given depth values

that computes the Discrete Fourier Transform of a sequence of equally spaced samples of a signal. The task graph corresponding to an input consisting of 4 data points is in Figure 3.6(a) [76]. Any path starting from the source task to any of the leaf tasks is a critical path. For an input of size m where $m = 2^k$ for some integer k, there are $2m - 1 + m \cdot log_2 m$ tasks in the graph. We have generated test cases for input sizes 2, 4, 8, 16 and 32. Figure 3.6(b) gives the outcome for 4 processors and CCR of 1 for all input sizes. CMA and BDSC produce similar makespans for smaller graphs, with CMA giving smaller makespans for larger graphs.

3) *Fork Join Graphs (FJ).* These graphs consist of a series of fork operations starting from a source node that are then followed by join operations to a leaf node. We use the depth and degree parameter to define these graphs. The depth specifies the number of recursive fork operations and degree specifies the number of children of a task produced during a fork. Figure 3.7(a) is a fork-join graph for a depth and degree of 2 [45]. Again, any path starting from the source task to the leaf task is a critical path. We generated test cases with degree in the range 2-4 and depth in 1-6. Figure 3.7(b) is the result for 5 processors, a CCR value of 5 and a degree of 3 for the different depth values. The difference in makespans for higher depths is more magnified than in the other test cases. This is because the size of the fork join graphs with depths of 4-6 are much larger than the largest graphs of the other test cases. Similar results are also seen for changing degrees. This shows that our algorithm gives higher gains for larger graphs with more communicating tasks.

4) *Molecular Dynamics Code (MDC).* Molecular dynamics is a method of computer simulation that is used to study physical movements of atoms and

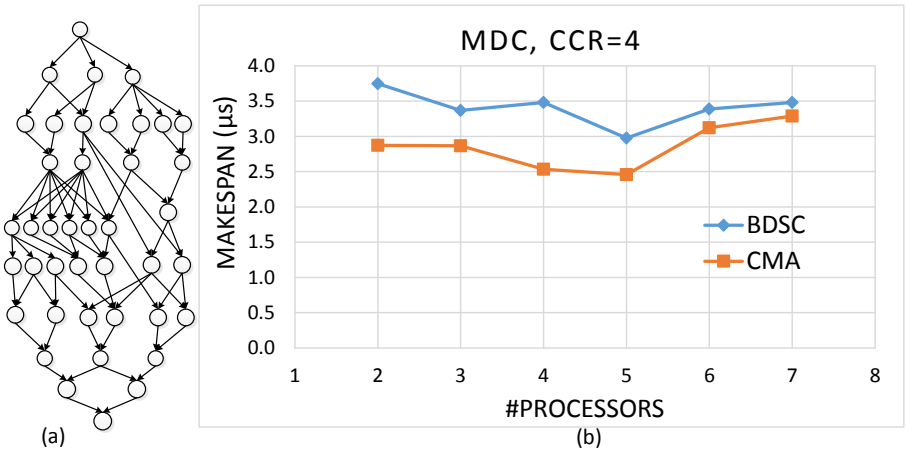


Figure 3.8: (a) MDC task graph, (b)MDC results over the number of processors

molecules. The graph of its code, in Figure 3.8(a) [76], has a fixed number of tasks and a relatively irregular structure compared to the other test cases which makes it also useful for evaluating our algorithm. We vary the CCR values and the number of processors (from 2 to 7) per test case. The outcome for CCR of 4 is given in Figure 3.8. CMA produces lower makespans than BDSC and we observe that the makespan difference reduces as we add resources, showing the benefit of using CMA for limited resources. Another interesting observation is that more processors does not necessarily imply smaller makespan. It is possible that with more processors the amount of synchronization needed among cores increases thus adversely effecting the makespan.

Table 3.4: Makespan comparison results of cma and bdsc for the well-known applications

Application	Lower m_{BDSC}	Lower m_{CMA}	Equal
GE	638	2185	57
FFT	77	537	286
FJ	373	2203	484
MDC	19	53	0

Table 3.4 summarizes the overall results of the above test cases. The first column lists the application type. The second and third columns give the number of test cases where BDSC and CMA respectively produce the lowest makespan. The last column gives the number of equal makespan cases. We see that CMA produces lower makespans than BDSC in a significantly higher number of test cases of each type. The box-plot of the percentage of improvements per case are given in Figure 3.9. The leftmost and rightmost whiskers give the lowest

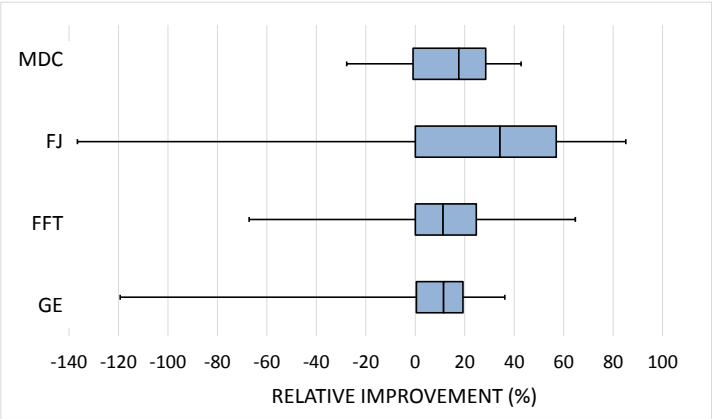


Figure 3.9: Box plots of relative percentage improvements of CMA over BDSC

and highest improvement respectively. The left and right ends of the box are the lower and upper quartiles and the line dividing the box is the median. The typical improvements lie in the range between the lower and upper quartile covering 50% of the cases. This range is 0.5-19% for GE, 0-25% for FFT, 0-57% for FJ and -0.9-28% for MDC which is mainly positive. To gain insight in the cases where BDSC results in better makespan, we have the box-plots of those cases in terms of number of processors, CCR and graph sizes in Figure 3.10. The graphs where BDSC performs better are typically smaller. This highlights the usefulness of CMA for larger graphs. Another observation is that the negative extremes (<-50%) for FJ and GE in Figure 3.9 are either small graphs with CCR of above 8 or bigger graphs bound to 2 processors. Both cases are outliers in Figure 3.10 implying that the negative extremes are due to a few corner cases that are not typical.

We also performed experiments by adapting CMA to message passing systems (using the *tlevel* computation of DSC) and observed that CMA and BDSC are similar in terms of the number of test cases where they perform best. This implies that CMA is also applicable to message passing systems. However, it confirms our hypothesis that the three-step approach of CMA is particularly beneficial when dealing with the inaccuracy in timing computations in the shared memory case.

3.6 Related work

Standard combinatorial optimization techniques such as branch and bound algorithms [23], tabu search [38] and simulated annealing [47] search the state-space for optimal solutions with different heuristic search strategies. To achieve good

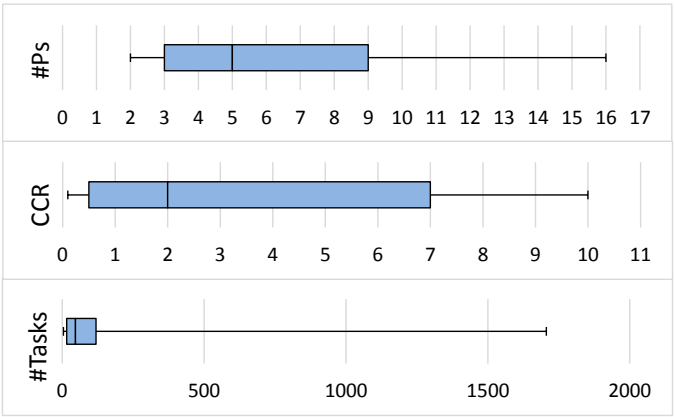


Figure 3.10: Box plot of (a) number of processors, (b) CCR values, and (c) number of tasks in the graph for test cases where CMA produces higher makespans

quality the algorithms rely on the exploration of a significant number of alternative solutions. This works very well for small to medium sized graphs in a reasonable amount of time. Our solution targets solutions for large DAGs constructing only a single solution. We therefore resort to list scheduling [41] and clustering [87] based methods that are efficient and commonly used in DAG binding and scheduling.

The clustering step in our work adopts the DSC algorithm in [87] and modifies it to deal with individual task deadlines and the shared memory communication mechanism. DSC has been shown to outperform other clustering methods for an unbounded number of processors in terms of results and algorithmic complexity [44]. DSC performs clustering assuming unlimited resources. We compare CMA to BDSC, an extension of DSC for limited resources, that uses a common binding method for both shared memory and message passing systems. It uses the justification that if it is worthwhile to nullify a communication cost for message passing systems, it is also worthwhile for shared memory systems. CMA, on the other hand, explicitly takes the shared memory communication framework into account in its extension of DSC that enables it to make better clustering decisions. Its three-step approach uses more high level structural information in the graph compared to the local decisions made by BDSC. The algorithmic complexity of BDSC is $O(|T|^3)$, which is the same as the complexity of CMA.

One of the earliest methods to consider platform size limitations, proposed by Sarkar in [64], performs clustering assuming unlimited resources and then uses an incremental approach to perform the assignment of clusters to resources iteratively. Each iteration selects and assigns a task along with all other tasks in the cluster to the processor that gives the least makespan increase from the last iteration. This needs makespan computation by scheduling all the tasks in the

cluster for each processor per iteration. Its complexity of $O(|T||R|(|T|+|D|))$ makes the algorithm slow for large graphs. A low complexity binding approach in PYRROS [86] first uses DSC to obtain the initial clusters. It then sorts the clusters in decreasing order of their loads and then maps them to the processors in a balanced manner. Such a mapping risks assigning independent clusters onto the same resource while separating dependent clusters onto different resources which can increase the makespan. CMA reduces this risk by performing an intermediate merging step based on dependencies. Also CMA, in its first step, adapts the clustering of DSC to deal with shared memory systems.

Another approach for limited processors, called Triplet [24], also adopts a three-step approach for binding to a system of heterogenous systems called workstations. The first clustering step on unlimited resources only uses task top levels to sort tasks. CMA utilizes more information by taking top levels and due-dates to sort tasks during clustering. After clustering, Triplet clusters the workstations to form homogenous workstation clusters. Lastly, clusters are sorted on the amount of external communication and workload and assigned to the workstation cluster that gives the earliest start time. In CMA's second step, we sort clusters on the number of dependencies with their highest connected cluster. We then merge the highest connected clusters together thus removing the communication cost between them. Only then we allocate the clusters to processors. This second step allows us to further reduce communication between clusters thus minimizing makespans.

An approach for shared memory systems and limited resources that considers the concept of synchronization time, like in CMA, is the Extended Latency Time (ELT) [70] approach. It calculates priorities of tasks using a combination of metrics including top levels and bottom levels. It then performs binding by choosing tasks based on the priority order, while keeping the priorities fixed. CMA, like DSC and BDSC, uses dynamic priorities which are updated during clustering to account for the edge costs that become zero when two tasks are clustered together. This helps in making better binding decisions as it can detect changes in the critical paths that can arise due to zeroing of edge costs, also pointed out in [49].

Other single-step binding algorithms include High Level First with Estimated Times [3], Heterogenous Earliest-Finish Time [76], and Constrained Earliest Finish Time [45]. In [44], BDSC is shown to outperform these as well as the earlier mentioned multi-step approaches. BDSC is thus chosen as our benchmark for comparison. This chapter shows that CMA outperforms BDSC for binding in shared memory systems.

Binding algorithms typically focus on makespan reductions and do not take task deadlines into account. An approach that does consider deadlines is given in [13]. However, it considers independent sporadic tasks and is preemptive also allowing task migration, none of which are applicable to our problem domain. CMA performs binding of DAGs on multiprocessor platforms by utilizing parallelism while taking both communication overhead and task deadlines into account.

3.7 Summary

This chapter contributes a three-step binding algorithm that computes the binding of tasks with deadlines on the limited resources of a shared memory multiprocessor platform. The first step, a deadline aware shared memory extension of the DSC algorithm, produces a clustered graph assuming unlimited resources. Since there can be more clusters than resources, the next step performs merging of clusters based on dependencies while constraining them to be smaller than a threshold. Finally the clusters are allocated to the resources by balancing the workload. CMA has been validated by evaluating its impact on our earliest due-date first scheduler on large control applications of ASML. CMA produces lower makespans in comparison to the state of the art BDSC algorithm. It also outperforms BDSC in a high number of test cases of other well-known parallel algorithms.

Chapter 4

Robustness analysis of static-order schedules

In the applications considered until now, tasks have fixed measured execution durations. Most of the time, tasks execute with execution times around these durations. However, as explained in Chapter 1, general purpose platforms are known to suffer from low predictability and applications running on them often exhibit fluctuations in execution timings. There is a need for a stochastic robust scheduler that can produce schedules that are tolerant to these variations and robust in nature. Due to the complexity of the problem, we revert back to the simpler task model where the binding of tasks is known and communication overhead is not considered. This chapter presents the first step towards robust scheduling by providing a means to define and measure robustness of tasks and static-order schedules. An early version of this work has been published in [7].

Robustness of a task is defined as the probability of meeting its deadline and the robustness of a schedule is expressed in terms of the expected value of the number of tasks missing their deadlines. Since nominal execution times of tasks are mostly closer to the best case execution time than to the worst case execution time, the probability density function of the task execution time is mostly not normally distributed but right skewed in nature. Multiprocessor schedules are characterized by dependencies that span along as well as across processors. Propagation of completion time distributions along these dependencies requires a combination of convolutions and maximization operations to be performed on the task execution and completion time distributions. Since there are no known practical analytical means to compute or accurately approximate the distribution

of the maximum of stochastic variables that are skewed in nature (e.g. skew-normal or PERT), we devise an approach that combines the advantages of both analytical computations and simulations to accurately estimate the robustness of tasks and schedules. Apart from being skewed, we have information on the bounds of the distributions in the form of best-case and worst-case task execution times. The combination of the skewness and the boundedness requirements is met by (modified) PERT distributions, as opposed to the skew-normal distributions which do not meet the boundedness requirement. Our robustness approach involves fitting a (modified) PERT distribution on the simulated results using analytically computed bounds. We present two possible ways of doing this curve fitting and compare them in the experimental section. The robustness analysis (and both its possible curve fitting techniques) has been tested for scalability on ASML applications. The chapter is further organized as follows. Section 4.1 gives the related work in robustness analysis. Section 4.2 defines the problem by giving the preliminaries followed by a problem description and a summary of the solution approach. Section 4.3 gives the major challenges of the analysis and Section 4.4 gives the details of the proposed practical realization of the analysis. Section 4.5 gives the experimental results and Section 4.6 summarizes the chapter.

4.1 Related work

In this section, we present literature survey pertaining to the three main aspects of this chapter, 1) different robustness measures presented in literature, 2) techniques to perform max-plus operations on distributions and 3) evaluation of the goodness of fit of a distribution on data samples.

First we elaborate on the existing work for analyzing robustness. A paper by Canon and Jeanot [22] does a survey of several robustness metrics for DAG schedules and compares them. It discusses a means to measure robustness, as presented in [10], by 1) defining what performance measure needs to be robust, 2) identifying what parameters impact its robustness, 3) identifying how changing these parameters effects robustness and 4) quantifying what amount of parameter variation causes loss in performance. Based on these steps, a metric called the 'robustness radius' is defined as the smallest variation of the parameter that affects the system performance. Such kind of definition does not take into account probabilities that some parameter changes occur more often than others. In our work, the performance measure is the number of deadline misses and the parameters that effects it are the variations in task execution times quantified using probability distributions. The authors of [32] use the Kolmogorov-Smirnov (KS) distance between the cumulative distribution of the performance measure with and without perturbations. A large distance implies that the perturbation has a big impact on robustness. In our work, the parameters that impact robustness are not external perturbations but the given execution time distributions of tasks. Other alternatives of quantifying robustness in literature are slack based metrics

as presented in [68]. An example is the slack mean defined as the mean of the slack in the schedule in [18]. These metrics are suitable when we do not have the stochastic execution time information that forms the basis of this work. In our work, we utilize this stochastic information to derive deadline miss probability-based metrics to quantify DAG schedule robustness.

There is also literature that does express robustness in probabilistic terms. In [83], the deadline miss probabilities of tasks running on unreliable hosts is estimated. They study the availability periods of hosts and their effect on tasks meeting their deadlines. In our work we study robustness of schedules in relation to task execution time variations that are known beforehand and internal to the tasks. Another probabilistic metric is presented in [67] that gives the probability that the makespan is within two bounds. This can be used to address the deadline problem by assuming the upper bound on the makespan to be the deadline that needs to be met. However, the approach only looks at independent applications and hence the propagation of distributions avoids the need to consider synchronization and hence the need to compute the maximum of stochastic variables. In our DAG schedules, tasks on different processors have data dependencies between them. When a task has dependencies from two source tasks on different processors, the start time of the tasks is the maximum of the completion times of the source tasks requiring the analysis of the maximum of stochastic completion times.

We will now look into the related work in performing the max-plus operations on distributions. The central limit theorem [61] states the convergence of many random variables (under certain fairly common conditions) to approximate a normal distribution under summation, but it does not apply to the maximum operator. In fact, it has been observed that the max of standard normal distributions can tend to be skewed depending on the difference in the standard deviations of the input terms. There are no known classes of continuous distributions that are closed under the maximum operation (such that the distribution of the maximum is also in the class). Work has been done to approximate the maximum of 2 or more standard normal random variables with another standard normal random variable up to a certain degree of accuracy such as presented in the work of Clark [25] and in [58, 15]. It has been shown in [58] that these approximations perform poorly when the standard deviations of the input distributions are different. In addition, we could not find methods of approximating the maximum of distributions that are skewed by nature. In the applications considered in the domain of this chapter, the distributions of the execution times of tasks are most often skewed (right skewed with nominal values closer to min). Hence, we cannot use normal approximations without losing accuracy in the results.

Alternatively, one could approximate distributions with a limited number of discrete values. However, this results in exponential computational complexity as all combinations for different tasks need to be enumerated [16]. Another approach to this computation is the enumeration of all the critical paths leading to a particular node. The propagation of distributions along these individual paths

follows the sum operator alone. Finally, maximization is applied to the distributions of all the critical paths. This approach separates the analysis into two parts: computing the timings of paths which can be computed with simple convolutions and which enjoy the central limit theorem and finally applying the maximization [8]. The drawback of this approach is the need to find the critical paths to be considered per node. For large schedules the number of critical paths that need to be detected can be very large.

Most approximation methods compare with extensive simulations to judge the accuracy of their approximations [73, 62]. This involves simulating a large number of samples to obtain the entire resultant distribution including the tails with often small probability mass that determine the deadline miss probabilities. This has the drawback of being too time consuming. In this work, we also use simulations, but we use the PERT distribution to shorten simulation times. The results of these quick simulations are combined with analytically computed bounds to obtain the distributions.

On the PERT fitting aspect, there exist distance measures in the literature that can be used to evaluate the goodness of fit of distributions on samples of data such as the KS-statistic in [32, 39]. These measures have their own probability distribution and studying these distributions gives an estimate on the closeness of the input distributions. The R-squared measure [21] used in regression analysis is a single value between 0 and 1 that indicates how well data fits the statistical model that is predicted using the analysis and is calculated using the sum of squares method on the data. In our work, we propose a new metric based on the inner product of the two distributions that also returns a single value between 0 and 1 which gives an estimate of their closeness. It can be used to evaluate the closeness of any two distributions and not just that of a distribution on data. One of our curve fitting approaches uses this metric in a divide and conquer search algorithm to accurately and with low complexity fit a PERT distribution on the histograms obtained from limited simulations.

4.2 Problem definition and solution overview

4.2.1 Preliminaries

In this chapter, the definition of a task is extended to replace the fixed variables corresponding to execution time, start time and completion time with stochastic variables. Additionally, since we revert back to the fixed binding assumption to focus on the robustness problem, we again constrain the binding of a task to be known by removing the $\perp$ value that was introduced in Chapter 3. We also revert back to the definition of a set of dependencies as the tuple $D \subseteq T^2$ that does not consider the communication cost.

Let $\mathbb{D}$ be the set of all probability density functions such that if $d \in \mathbb{D}$, then $d : \mathbb{R}^{\geq 0} \rightarrow \mathbb{R}^{\geq 0}$. For a DAG $G = (T, D)$, a *task* $a \in T$ is a tuple $a = (P_{e_a}, r_a, d_a) \in$

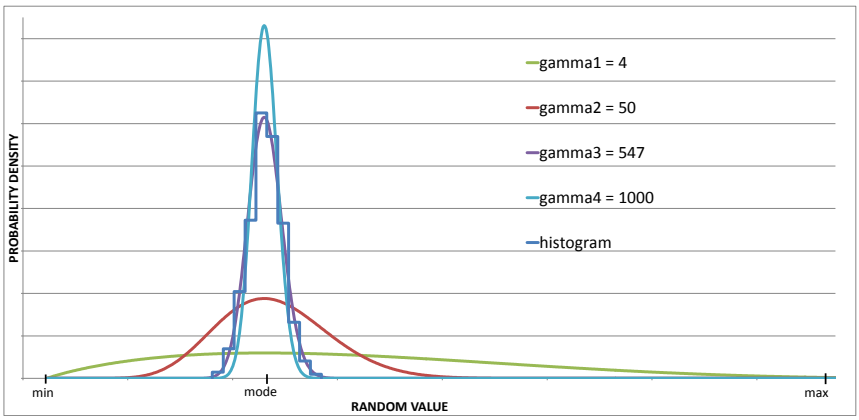


Figure 4.1: Modified PERT distributions with different γ values.

$\mathbb{D} \times R \times \mathbb{R}^{\geq 0}$. Here, $r_a \in R$ is the resource that a is bound to from the set of resources R, P_{e_a} is the probability density function of the continuous execution time distribution of a, with e_a denoting the random variable for the execution time of a, and d_a is the deadline of a.

We use the fixed binding and assume a given execution order of tasks in a static-order schedule to perform robustness analysis of run-time scheduling. Given a static-order schedule, the tasks have stochastic start and completion times due to their random execution times. For task a, s_a^S and c_a^S denote the random variables for the start and completion time in S, respectively. The start time of the first task scheduled on a resource, without any predecessors, is zero with probability 1. Completion times are derived by adding the task execution times to the corresponding task start times. The probability density functions for the start time and the completion time of a are denoted by $P_{s_a}^S$ and $P_{c_a}^S$ respectively.

A *PERT distribution* [82] is a version of the Beta distribution and is defined by three parameters, namely the minimum (*min*), the most likely value (*mode*) and the maximum (*max*). It derives its name from the PERT (project evaluation and review technique) networks, a statistical tool used in project management to analyse, with respect to their timings, the tasks involved in completing a project.

A *modified PERT distribution* is a variant of the PERT distribution developed by David Vose and allows producing shapes with varying degrees of uncertainty by means of a fourth parameter, gamma (γ), that scales the width (variance) of the distribution. In the standard PERT, $\gamma = 4$ and upon increasing the value of γ, the distribution becomes more concentrated around the mode. On the other hand on decreasing the value of γ, the distribution gets more spread out between the *min* and *max*. Figure 4.1 shows the standard PERT distribution with $\gamma = 4$

and three example modified PERT distributions for higher values of γ together with their fitting on a histogram for a particular task in the ASML schedules. The probability density function for the modified PERT distribution with *min*, *max*, *mode* and γ as parameters is given by the following equation [78]:

$$P(x) = \begin{cases} \frac{(x-min)^{\alpha_1-1}(max-x)^{\alpha_2-1}}{\beta(\alpha_1,\alpha_2)(max-min)^{\alpha_1+\alpha_2-1}} & min \leq x \leq max \\ 0 & \text{otherwise} \end{cases}$$

where the shape parameters α_1 and α_2 are given in Equation 4.1 and the beta function ($\beta(\alpha_1, \alpha_2)$) is given in Equation 4.2.

$$\alpha_1 = 1 + \gamma\left(\frac{mode - min}{max - min}\right); \alpha_2 = 1 + \gamma\left(\frac{max - mode}{max - min}\right) \tag{4.1}$$

$$\beta(\alpha_1, \alpha_2) = \int_0^1 t^{\alpha_1-1}(1-t)^{\alpha_2-1}dt \tag{4.2}$$

4.2.2 Problem description

A DAG schedule is a fixed static-order and binding of tasks on a multiprocessor. Tasks in a schedule have dependencies between them both along and across processors. We make two additional assumptions: (1) *Task execution time distributions are known and have finite support, i.e., lower and upper bounds outside of which the probability density is zero.* A deadline is given for each task. (2) *Execution times of tasks are independent.* In reality positive or negative correlations between execution times of tasks may exist, but the focus of this work is to study how robustness of schedules is influenced by scheduled order and synchronization of the tasks.

Robustness is a measure of the tolerance of a task or schedule to variations in the execution times of tasks. Tolerance is measured by the probability that the tasks in the schedule still meet their deadlines in the presence of these variations. First, robustness of tasks to missing their deadlines is defined and from that robustness of the schedule as a whole is defined. The statement of the problem being dealt with in this chapter is:

Problem Statement 3. *Given static-order schedule S, what is the robustness of S and its constituent tasks?*

To obtain deadline miss probabilities, we need to derive task completion time distributions from task execution time distributions and the given schedule. There are two obvious approaches to doing this: (1) compute the completion time distributions analytically, (2) use the execution time distributions to draw execution time samples and perform extensive simulations to obtain the full completion time distributions. However, computing the completion time distributions analytically is highly complex due to the presence of the max operations owing to the dependencies between the tasks. This will be further elaborated in Section 4.3. On

the other hand performing simulations alone will require us to perform extensive, time consuming simulations to be able to derive full completion time distributions with sufficiently accurate tails to estimate deadline miss probabilities. Instead, we use an approach which combines limited simulations with analytically computed bounds to estimate the completion time distributions.

4.2.3 Solution flow

In this subsection we summarize the overall approach of the chapter. Deadline miss probabilities can be derived from the distributions of the completion times of the tasks and their deadlines. Before we can compute the completion time distributions, we need task execution time distributions. These are typically approximated using statistical data from measurements, as shown in Figure 4.2(a). We present two curve fitting techniques that can be used to fit a PERT distribution on measurement samples. Taking the lower bound (min) and upper bound (max) of the execution times, one technique uses a nested divide and conquer search on the histograms of the measurement samples to obtain the unknown $mode$ and γ PERT parameters. The other technique directly computes the $mode$ and γ from the min and max together with the mean and variance of the measurement samples. We compare the two fitting techniques in the experimental section. Once we have the execution time distributions, we need to derive the completion time distributions of the tasks. Due to the difficulties of performing max and plus operations on distributions [16], this cannot be done entirely analytically. On the other hand, performing only simulations produces insufficient rare deadline misses depending on the length of the simulations as shown in Figure 4.2(b) and (c). Hence, we instead approximate the completion time distributions as PERT distributions using analytically computed min and max and data from limited simulations, carried out by drawing samples from the given PERT execution time distributions. This is shown in Figure 4.2(d). For this approximation, we consider the same two mentioned curve fitting techniques. Once we have obtained the estimates of the completion time distributions, we can calculate task robustness as the red shaded portion of the area under the density function in Figure 4.2(e). Schedule robustness is then quantified as the expected number of task deadline misses. Although not a focus of this work, it might be possible to adapt the fitting algorithms to apply the overall approach even when the distributions are not PERT-like.

4.3 Challenges of the analysis

In this section we explain in some detail the analytical model and the reasons for its complexity. This is followed by an explanation on why an approach using only simulations is also not practically feasible.

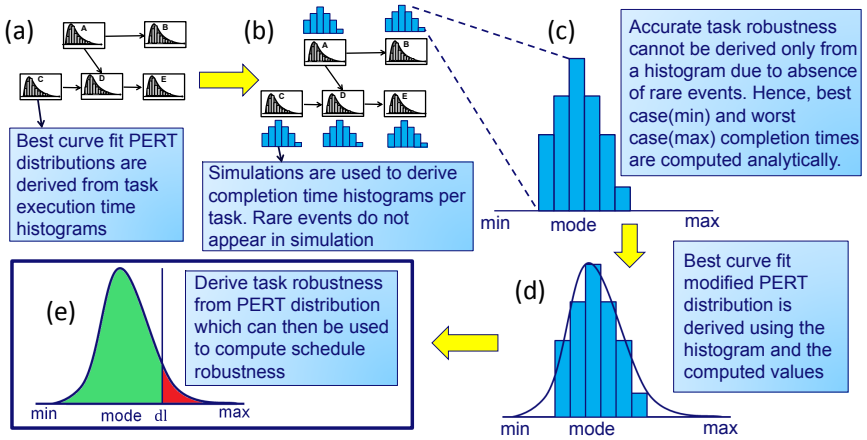


Figure 4.2: Robustness analysis: flow of solution approach.

4.3.1 Analytical approach only

Since we want to compute the deadline miss probabilities of all tasks in the schedule, we need to compute and propagate the completion time distributions per task. Computing these distributions under the maximum operation is very difficult as is explained below. Given the start and execution time distribution of a task a in a schedule S, the completion time distribution is the distribution of their sum and is computed as follows.

$$P^S_{c_a}(t) = \int_0^\infty P^S_{s_a}(t') \cdot P_{e_a}(t - t')dt' \tag{4.3}$$

The start time of a task without predecessors, when there is no task scheduled on its resource r is 0, and its distribution is as follows ($\delta(t)$ represents the Dirac δ-function).

$$P^S_{s_a}(t) = \delta(t) \tag{4.4}$$

The start time of a task which has no predecessors but has tasks scheduled before it on its resource r, with $before(r, a)$ being the last task scheduled on r before a, is the completion time of $before(r, a)$.

$$P^S_{s_a}(t) = P^S_{c_{before(r,a)}}(t) \tag{4.5}$$

The start time of a task a with predecessors is the maximum of the completion time of the last completing predecessor ($lastPred(a)$) and that of $before(r, a)$.

$$s_a = \max(c^S_{before(r,a)}, c^S_{lastPred(a)}) \tag{4.6}$$

The corresponding start time distribution is computed as follows.

$$\begin{aligned} P^S_{s_a}(t) = P^S_{c_{before(r,a)}}(t) \int_t^\infty P^S_{c_{lastPred(a)}}(t')dt' \\ + P^S_{c_{lastPred(a)}}(t) \int_t^\infty P^S_{c_{before(r)}}(t')dt' \end{aligned} \tag{4.7}$$

If the completion times of $lastPred(a)$ and $before(r,a)$ are dependent, due to the existence of dependencies to common tasks, one would have to resort to computing it from the joint distribution of $lastPred(a)$ and $before(r,a)$.

$$\begin{aligned} P^S_{s_a}(t) = \int_t^\infty P^S_{c_{before(r,a),lastPred(a)}}(t,t')dt' \\ + \int_t^\infty P^S_{c_{lastPred(a),before(r,a)}}(t',t)dt' \end{aligned} \tag{4.8}$$

Such a joint distribution also includes the information about the correlation between the individual completion time values of the tasks. As such, obtaining this distribution is difficult in practice. The completion time distribution can be computed from the start time distributions using Equation 4.3. However, obtaining the start time distributions is hard even without correlations. This is because there are no continuous distributions that are known to be closed under the max operation, which captures synchronization on input dependencies and execution times. Hence, even if $P^S_{c_{before(r,a)}}$ and $P^S_{c_{lastPred(a)}}$ are known distributions, the distribution of their max need not be the same or even a known distribution. Also, if $P^S_{c_{before(r,a)}}$ and $P^S_{c_{lastPred(a)}}$ are the max of certain other distributions (from earlier in the schedule) then it is possible that their properties are already not known. Due to this it becomes very hard to compute their integrals. Alternatively we could consider discrete enumerations of $P^S_{c_{before(r,a)}}$ and $P^S_{c_{lastPred(a)}}$ for the computation. We would then need to compute the max (or sum) for each combination of discrete values from them. The complexity of this computation grows exponentially in the size of the task graph and the number of possible discrete values [71]. This is clear when we consider a schedule with n tasks and each task has m possible values for its execution time. The number of possible values for the completion time is m^n. Industrial schedules may have thousands of tasks executing on general purpose platforms and exhibiting large variations in their execution timings. To avoid the exponential complexity of such a discrete approximation, we should instead be able to somehow approximate the resultant

distribution to some known distribution with parameters computed in terms of the parameters of the input distributions. However, as already seen in Section 4.1, there are also no known analytical approximations for the parameters of the max of skewed distributions. As such, we could not compute the completion time distributions of tasks analytically alone.

4.3.2 Simulations only

Simulations are performed on the schedules by drawing samples from the task execution time distributions. Extensive simulations to obtain a large number of completion time samples can be used to estimate the entire completion time distributions. The advantage of this approach is that we do not need to keep track of the correlations between the various distributions due to dependencies between tasks since they are inherently carried across in the simulations. The main drawback is that extensive simulations require a significant amount of time. In particular, simulating fewer samples results in the drawback that events with very low probability of occurrence (such as deadline misses often are) may not appear at all or too infrequently to accurately estimate their likelihood. As a result, with limited simulations we only obtain values around the most likely completion times and miss out those that are less likely. In this scenario, we will find the probability of missing deadlines to be estimated very inaccurately. Hence, we need an approach that combines the accuracy of the analytical approach to obtain rare events with the strength of simulations to generate mass around the most likely events and to naturally handle correlations due to task dependencies, to obtain completion time distributions.

4.4 Proposed robustness analysis approach

We assume that task execution time distributions are known apriori. Mostly in reality, the information known about task execution times is limited to measurements. PERT distributions are fitted on the measurements to obtain execution time distributions. Later on in Section 4.4.4, PERT distributions are fitted on completion time samples from simulations in combination with analytically computed bounds to obtain completion time distributions. We present two approaches to fit PERT distributions on samples (measured samples for execution times and simulated samples for completion times). The parameters of a PERT distribution are *min*, *max*, *mode* and γ. Along with the samples, both approaches needs the *min* and *max* of the distributions. In case of the execution time distributions, the *min* and *max* are set to be the lower bound and the upper bound values obtained from the measurements. For the completion time distributions, we use analytically computed bounds as *min* and *max* which are further elaborated in Section 4.4.4. We compute these bounds analytically since they can be efficiently computed and the simulations are too limited to obtain these extreme values as

already explained in Section 4.3.2. To evaluate the fitting approaches, we first define a metric that quantifies how well a PERT distribution fits on a histogram. One of the curve fitting techniques also uses this metric in its fitting process.

4.4.1 Curve fitting metric

A histogram is a means of categorizing data samples into a number of bins. These bins are identified by a specific range or bin-interval and the width of the interval is the bin-width represented as Δ. In this chapter, we consider all bins of a histogram to be of the same width. The normalized height of the bin is given by the number of elements falling within the bin interval divided by the total number of elements in the histogram. We obtain the frequency density by dividing the height of the bin with the bin-width. The number b of bins is a parameter of our approach. Depending on the number of samples, the value of b must be chosen taking into account the trade-off between the fine-graininess of the bin-intervals and the irregularities in the resultant bin heights due to the limited statistical information. In order to define the best fit of one distribution on another, we need a metric that quantifies the fit between two distributions.

Definition 6. (L^2 FUNCTIONS) *A function $f(x)$ is said to be square integrable if $|f|^2 = \int_{-\infty}^{\infty} f(x)^2 dx$ is finite. An L^2 function is a function that is square integrable. In such a case $|f|$ is called its L^2-norm.*

Definition 7. (L^2 INNER PRODUCT) *Given two L^2 functions f and g, their inner product is given by*

$$\langle f, g \rangle = \int_{-\infty}^{\infty} f(x) \cdot g(x)\ dx, \tag{4.9}$$

Note that $|f|^2 = \langle f, f \rangle$.

Based on the above definitions, we define the curve fitting metric using the normalized inner product of the PERT and the histogram as follows:

Definition 8. (CURVE FITTING METRIC) *Given a PERT distribution p and a histogram h, both L^2 functions, their curve fitting metric ($M_{\langle p,h \rangle}$) is defined by*

$$M_{\langle p,h \rangle} = \frac{\langle p, h \rangle}{|p| \cdot |h|} \tag{4.10}$$

We normalize the inner product with the L^2 norms of the PERT and the histogram. This is to scale their respective lengths in order to obtain a metric in the range [0,1] that describes how well a PERT curve fits on a histogram. Since the histogram is discrete with a fixed number of bins b, each density value can be obtained using the rectangular function. We use δ_Δ to denote a rectangular function over a bin width Δ and height $\frac{1}{\Delta}$, centered from 0 to Δ.

$$h(x) = \sum_{1 \leq k \leq b} h_k \cdot \delta_\Delta(x - l_k) \tag{4.11}$$

where l_k is the left boundary of the k^{th} bin and h_k is its normalized height obtained by dividing the number of the elements in the bin with the total number of elements in the histogram. Given this, the inner product of h and the continuous PERT distribution p is computed as follows.

$$\begin{aligned} \langle p, h \rangle &= \int_{-\infty}^{\infty} p(x) \sum_{1 \leq k \leq b} h_k \cdot \delta_\Delta(x - l_k) dx \\ &= \frac{1}{\Delta} \sum_{1 \leq k \leq b} h_k \int_{l_k}^{r_k} p(x) dx, \end{aligned} \tag{4.12}$$

where l_k and r_k are left and right boundaries of the k^{th} bin. To obtain the curve fitting metric of Equation 4.10, we divide the inner product of p and h by both their L^2 norms obtained by taking the square root of their respective L^2 inner products. Using Definition 7 and Equation 4.11, the L^2 inner product of the discrete histogram h with itself can be computed.

$$\begin{aligned} \langle h, h \rangle &= \int_{-\infty}^{\infty} \sum_{1 \leq k \leq b} h_k \cdot \delta_\Delta(x - l_k) \cdot \sum_{1 \leq k \leq b} h_k \cdot \delta_\Delta(x - l_k) dx \\ &= \frac{1}{\Delta^2} \sum_{1 \leq k \leq b} h_k \cdot h_k \cdot \Delta = \frac{1}{\Delta} \sum_{1 \leq k \leq b} h_k^2 \end{aligned} \tag{4.13}$$

where Δ is the bin-width of the bins of the histogram.

The L^2 inner product of p with itself can be reduced to the following expression from the PERT equations in Section 4.2.1 (we used Mathematica for the reduction).

$$\langle p, p \rangle = \frac{\Gamma(2\alpha_1 - 1) \cdot \Gamma(\alpha_1 + \alpha_2)^2 \cdot \Gamma(2\alpha_2 - 1)}{(max - min) \cdot \Gamma(\alpha_1)^2 \cdot \Gamma(\alpha_2)^2 \cdot \Gamma(2(\alpha_1 + \alpha_2 - 1))}, \tag{4.14}$$

where α_1 and α_2 are obtained using Equation 4.1 and $\Gamma(n)$ is the Gamma function [11] on n. The following two subsections explain our approaches to fit a PERT distribution on execution or completion time histograms.

4.4.2 Curve fitting using divide and conquer search for best fit

The first curve fitting technique fits a PERT distribution onto a histogram. For execution times, the measured discrete values can be classified into histograms.

Algorithm 7: DCS_m

Input : $m_{low}, m_{high}, \gamma_{low}, \gamma_{high}, prec_m, prec_\gamma$
Output: Best fit m, Best fit γ, Best fit M

$m_1 := m_{low}$;
$m_4 := m_{high}$;
$interval := \frac{(m_4 - m_1)}{3}$;
while $interval \geq prec_m$ **do**
  $m_2 := m_1 + interval$;
  $m_3 := m_1 + 2 * interval$;
  for *each* $x \in \{1, 2, 3, 4\}$ **do**
    $(\gamma_x, M(m_x)) := DCS_\gamma(m_x, \gamma_{low}, \gamma_{high}, prec_\gamma)$
  end
  $[m_1, m_4] := selectSegment(M(m_1, \gamma_1), M(m_2, \gamma_2), M(m_3, \gamma_3), M(m_4, \gamma_4))$;
  $interval := \frac{(m_4 - m_1)}{3}$;
end
return $(m_1, \gamma_1, M(m_1, \gamma_1))$

For completion times, the simulation samples are classified into histograms. With the information of the *min* and *max* of the distributions and these histograms, the aim of this curve fitting technique is to derive the *mode* and γ for the PERT distribution with the best fit on the histogram (giving the highest value for the curve fitting metric).

We use a divide & conquer search (DCS) approach given in Algorithm 7. The algorithm efficiently searches for a local maximum in a function of two variables on a given interval. The algorithm is used with the function that returns the metric $M_{\langle p,h \rangle}$ for a given histogram h and PERT distribution p with parameters m and γ, where m and γ are the *mode* and γ parameters. The search converges quickly without requiring a search through all the points in the search space. To compute the best *mode* and γ combination, a nested divide & conquer search is applied. At the top level, the algorithm searches for the optimal *mode*. To compare different modes, we need to compute the value of the metric corresponding to this *mode* with its optimal value for γ. So for each chosen *mode*, a second level of divide & conquer search is applied to look for the optimal γ, as given in Algorithm 8. Algorithm 7 takes as input the range for the *mode* values ($[m_{low}, m_{high}]$), the range for the γ values ($[\gamma_{low}, \gamma_{high}]$) and the precision up to which we continue the search for the optimal *mode* and γ denoted as $prec_m$ and $prec_\gamma$. The lower and upper bounds on the *mode* are the left boundary of the first bin and the right boundary of the last bin, respectively. The lower bound on γ is 4 (default value γ for PERT distributions) and the upper bound is chosen to be a sufficiently large value (10^4 here) that does not exclude the optimum, based on experiments.

Algorithm 8: DCS_{γ}

Input : $m, \gamma_{low}, \gamma_{high}, prec_{\gamma}$
Output: Best fit γ, Best fit M

$\gamma_1 := \gamma_{low}$;
$\gamma_4 := \gamma_{high}$;
$interval := \frac{(\gamma_4 - \gamma_1)}{3}$;
while $interval \geq prec_{\gamma}$ **do**
    $\gamma_2 := \gamma_1 + interval$;
    $\gamma_3 := \gamma_1 + 2 * interval$;
    for *each* $x \in \{1, 2, 3, 4\}$ **do**
        Compute $M(\gamma_x)$ using γ_x and the fixed m
    end
    $[\gamma_1, \gamma_4] := selectSegment(M(m, \gamma_1), M(m, \gamma_2), M(m, \gamma_3), M(m, \gamma_4))$;
    $interval := \frac{(\gamma_4 - \gamma_1)}{3}$;
end
return $(\gamma_1, M(m, \gamma_1))$

Algorithm 7 works by choosing four equidistant *mode* points (m_1, m_2, m_3 and m_4) covering the given range. When the distance between these points (*interval*) is below the precision $prec_m$ the while loop exits. For each of these four points, the optimal γ and corresponding metric value (M) are computed using Algorithm 8. Algorithm 8 similarly employs a search on the γ parameter by choosing four equidistant γ points in the range $[\gamma_{low}, \gamma_{high}]$. At each of these γ points, with given *mode*, *min* and *max*, the value of the metric is computed with the formulae for the curve fitting metric given in Equation 4.10.

In both algorithms, based on the values of M at these (*mode* or γ) points, the *selectSegment* function updates the left and right most points by eliminating at least one and sometimes two of the three intervals based on the values of the metric at each of the points and decides which segment(s) can contain the local maximum assuming uni-modal behavior of the function. For instance, consider Figure 4.3 which shows a mapping of points (*mode* or γ) to M values such that $M_1 < M_2$, $M_2 > M_3$ and $M_3 > M_4$. This indicates that the curve is increasing between M_1 and M_2 and decreasing between M_2 and M_3 and continues to decrease until M_4. As such the peak cannot lie between M_3 and M_4. The interval is reduced to $[M_1, M_3]$. By continuing the search in this manner, the interval between the points is reduced in every iteration of the loop to at most $2/3^{rd}$ and possibly even $1/3^{rd}$ of its original size. When the while loop exits the search returns the highest metric and the corresponding γ (for Algorithm 8) or *mode* and γ combination (for Algorithm 7) from the last four points. This algorithm has logarithmic complexity in the size of the interval and converges quickly. It is possible that the histogram being fitted using the DCS algorithm has multiple (local or global) peaks. In this case the algorithm tends to fit a PERT distribution with a *mode* either around

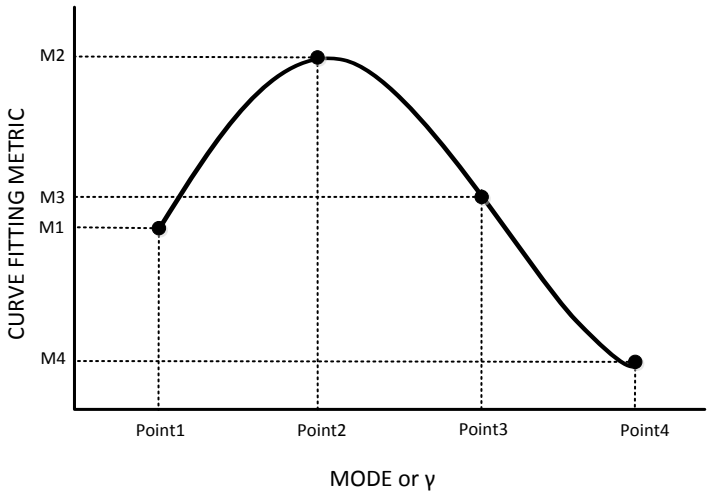


Figure 4.3: Divide and conquer search using four equidistant points.

one of the peaks or somewhere in between in such a manner that the overall curve fit is good. The speed of the divide and conquer is preferred over the accuracy of a full search. With respect to γ, we expect the function to be uni-modal; a proof of this conjecture remains to be done.

4.4.3 Curve fitting using PERT equations

A faster alternative to the divide and conquer search presented in the previous section is to use PERT equations to compute *mode* and γ directly from the *min*, *max* and execution time or completion time samples. We do not need classification of samples into histograms for this method. We first compute the arithmetic mean and variance of the samples, denoted by μ and *var* respectively. The *mode* and γ parameters are then computed from the *min*, *max*, μ and *var* as follows:

$$\gamma = \frac{(\mu - min) \cdot (max - \mu) - 3 \cdot var}{var} \tag{4.15}$$

$$mode = \frac{\mu \cdot (\gamma + 2) - min - max}{\gamma} \tag{4.16}$$

Equations 4.15 and 4.16 were obtained by rearranging the expressions defining the *mean* and *var* of a modified PERT distribution given below [78].

$$\mu = \frac{min + \gamma \cdot mode + max}{\gamma + 2} \tag{4.17}$$

$$var = \frac{(\mu - min) \cdot (max - \mu)}{\gamma + 3} \tag{4.18}$$

This alternative is much faster than the divide and conquer search. In the experimental section, we use the curve fitting metric from Section 4.4.1 and the robustness metric that is presented in Section 4.4.5 to compare the two curve fitting techniques. The best approach is used within the robust scheduler of Chapter 5.

4.4.4 Obtaining completion time distributions: Combining analysis and simulations

To obtain the completion time distributions, we first analytically compute the *min* and *max* points of the PERT distributions from the bounds of the execution time distributions. Since the schedules are static-order, the best case and worst case completion times can be computed by considering all tasks in their best case and worst case execution times, respectively. In the static-order schedule, the fixed execution order of tasks on processors and the processor to processor synchronization mechanism are monotone, i.e., when a task execution time increases, completion times of tasks that follow it in the schedule either remain the same or increase also. Hence, worst-case execution times of tasks are known to lead to worst-case completion times of other tasks and similarly for best-case. As a result, there cannot be any scheduling anomalies allowing for a straightforward computation of the bounds on the completion times of tasks. Once we have the *min* and *max*, we draw samples from the PERT execution time distributions and perform simulations to obtain completion time samples which are converted to histograms. Each simulation corresponds to one run through the graph and returns one completion time sample per task. Since we generate samples by walking through the tasks and dependencies in the graph, correlations due to data dependencies are implicitly taken into account in these completion time samples. We then compute the *mode* and γ values using one of the two curve fitting techniques presented in Section 4.4.2 and Section 4.4.3 to obtain completion time distributions. Note that the required number of simulations is relatively small, because we only need to estimate the PERT parameters rather than the full distribution, as demonstrated in the experiments section.

4.4.5 Robustness metrics

The robustness of a task can be measured in terms of the probability of missing its deadline. Given the distribution for the completion times of a task a, the probability of missing its deadline d_a is computed as follows.

$$\mathbb{P}[c_a^S > d_a] = \int_{d_a}^{\infty} P_{c_a}^S(t)dt \tag{4.19}$$

Given the modified PERT completion time distribution of a task a and its deadline d_a, its probability of deadline miss is the cumulative probability density of the PERT from d_a to max. It can be computed by substituting the PERT equations from Section 4.2.1 to Equation 4.19 to obtain the following expression.

$$\mathbb{P}[c_a^S > d_a] = \int_{d_a}^{max} \frac{(t - min)^{\alpha_1 - 1}(max - t)^{\alpha_2 - 1}}{\beta(\alpha_1, \alpha_2)(max - min)^{\alpha_1 + \alpha_2 - 1}} dt \tag{4.20}$$

Given the probabilities of deadline misses per task, we define a random variable X to express the number of tasks that miss their deadline in a schedule. The probability distribution for this random variable is a discrete distribution with probability values for any x tasks missing their deadlines.

$$P(X = x) : \text{Probability that } x \text{ tasks miss their deadlines} \tag{4.21}$$

The expected value of this random variable gives the expected value of the number of tasks that miss their deadlines in a schedule S. It is one metric that quantifies the robustness of S and can be derived by taking the sum of the deadline miss probabilities of its constituent tasks. Note that this also applies if the completion time distributions of the tasks are dependent. We choose to measure robustness using the expected number of tasks missing deadlines instead of meeting deadlines for typically being the smaller number of the two. A highly robust schedule will thus have zero expected number of deadline misses.

$$E[X] = \sum_{a \in T} \mathbb{P}[c_a^S > d_a] \tag{4.22}$$

Normalizing the expected value with the number of tasks in S gives a metric that can be used to compare schedules of different sizes.

$$\overline{E[X]} = \frac{\sum_{a \in T} \mathbb{P}[c_a^S > d_a]}{|T|} \tag{4.23}$$

The following section gives the experimental results obtained by applying this approach on real schedules.

4.5 Experimental results

We performed robustness analysis on the schedules of three applications of the TWINSCAN NXE 3300B version of the wafer scanners, namely (1) *Stages*, (2) *Wafer handler* and (3) *POB*. Stages and POB applications have been introduced in previous chapters. Wafer handler is an application that performs multiple tasks to manoeuvre the wafer into the wafer stage to be exposed while regulating its features such as temperature, positioning and orientation as required.

The schedule of this stages application consists of 4219 tasks running at a frequency of 10kHz on a platform consisting of 12 general purpose single-core processors. The deadlines assigned to non-critical tasks is 100% of the processor budget. We instead analyzed the actual schedules on 80% of the processor budget, targeting the analysis to future high performance machines. This gives the tasks a deadline of $8 \cdot 10^{-5}s$. There are also 22 critical tasks that are assigned deadlines ranging between $4 \cdot 10^{-5}$ to $4.5 \cdot 10^{-5}s$. The handler schedule has 1681 tasks running at 10kHz on a platform with 3 general purpose single-core processors. The POB schedule has 2442 tasks running at 10kHz on a platform of 6 general purpose single-core processors. Task deadlines assigned are the same as for the stages schedule.

We computed PERT execution time distributions for tasks, based on available measurement statistics. We then performed robustness analysis by drawing samples from these distributions, simulating completion time histograms, and fitting PERT distributions with analytically computed bounds. The simulations were performed by converting the schedule models to discrete event simulation models in POOSL [81] and run with the Rotalumis simulation tool. We chose to present the results of the stages schedule in more detail compared to the other two schedules due to bigger size and higher complexity. The handler and POB schedules produce similar results and so we only list them in a table and leave out the details.

4.5.1 Evaluation of the robustness analysis approach

To perform robustness analysis, we simulated 1000 samples of the stages schedule and performed curve fitting using the two approaches presented in Section 4.4. For the nested divide and conquer search, we chose $prec_\gamma$ to be 1 and $prec_m$ to be the interval obtained by dividing the x-axis of the histogram considered into 100 parts. The number of bins in the histograms was chosen to be 10. To visualize how the PERT fitting of the two approaches is able to approximate mass around the rare events, we show the PERT fit on the completion time histograms of two tasks of the stages application. Figure 4.4(a) shows a task with sufficient mass in the histogram to produce an accurate fit. On the other hand Figure 4.4(b) shows a task with missing mass around the rare events which is covered by the PERT distributions. We see that both the PERT fitting approaches produce a good curve fit in either case. To draw comparisons, we plot the value of the curve fitting metric obtained for all tasks of the stages schedule using both techniques in Figure 4.5. The tasks on the x-axis are sorted on the values of the curve fitting metrics obtained using the DCS method. We see that all tasks have a good curve fit value M of above 93% from both techniques. This shows that PERT is a good distribution for the completion times. DCS performs slightly better with an average M value of 0.9895 in comparison to 0.9889 from the PERT equations. This was also observed on the handler and the POB schedules. The robustness of the stages schedule, in terms of the expected value of the number of tasks that

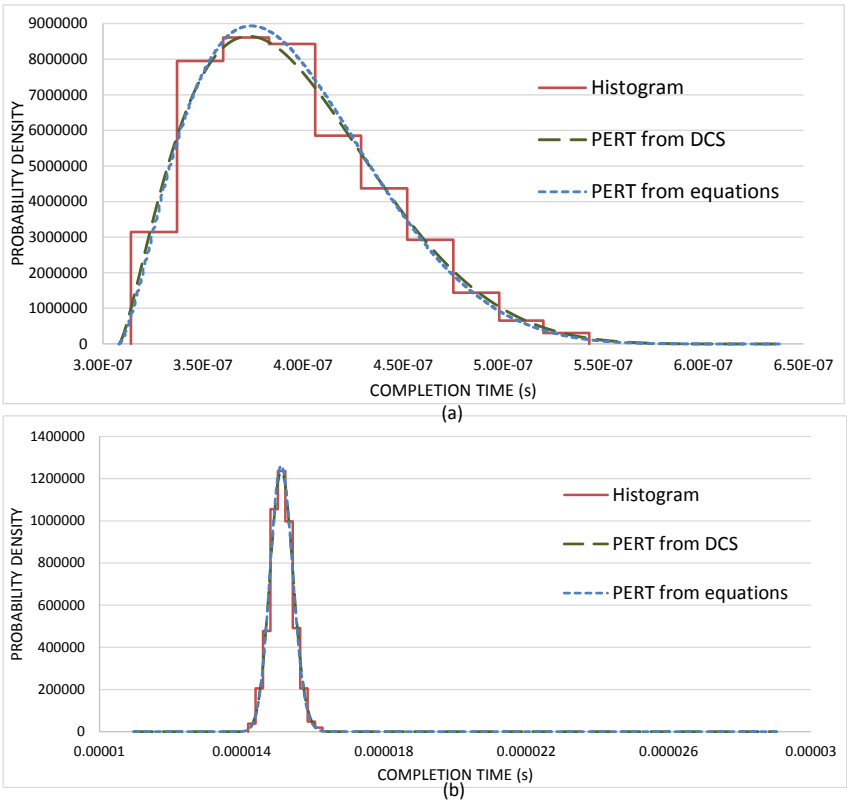


Figure 4.4: (a) PERT fit on a histogram with sufficient mass at the extremes and (b) PERT fit on a histogram with missing rare events.

miss their deadlines from Equation 4.22, was found to be around 79.02 from DCS and 78.80 from PERT equations (which is around 2% of the tasks). The average of the number of tasks that miss deadlines in all the 1000 simulation runs was found to be around 78.80 with a variance of around 138 and standard deviation of around 11, which is close to our predictions.

Table 4.1 lists the results for all the three schedules. The first column gives the name of the schedule and the second column gives the number of tasks. The third column gives the average number of deadline misses from the 1000 simulations. The fourth and fifth columns give the expected number of deadline misses obtained from the robustness analysis using the two curve fitting techniques. 'Eqn' refers to the curve fitting approach using PERT equations. We see that both techniques are quite accurate with respect to the simulation results. PERT equations performs

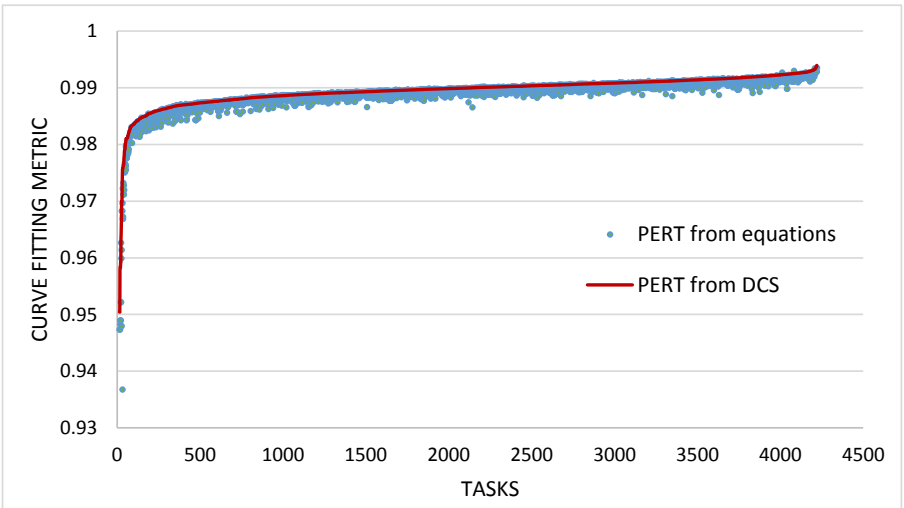


Figure 4.5: Curve fitting metrics for all tasks of the TWINSCAN NXE stages application.

Table 4.1: Robustness analysis outcome on the TWINSCAN NXE applications

Sch	#Tasks	Avg sim d misses	E[X]		Avg M		Runtime(s)		Sim time(s)
			DCS	Eqn	DCS	Eqn	DCS	Eqn	
Stages	4219	78.80	79.02	78.80	0.9895	0.9889	908	76	70
Handler	1681	36.55	36.68	36.54	0.9902	0.9897	372	30	27
POB	2442	14.48	14.48	14.46	0.9901	0.9896	533	44	40

slightly better for the stages and handler schedules, whereas DCS performs slightly better for the POB schedule. The sixth and seventh columns give the average value of the curve fitting metric from the two techniques. DCS produces better average curve fit in all three schedules. Column eight and nine give the run time of the robustness analysis using the two techniques which also includes the simulation time. Lastly, the tenth column gives the time taken for the simulations alone. We see that DCS takes significantly higher amount of time compared to PERT equations.

4.5.2 Validation with extensive (day-long) simulations

In order to validate our results we performed extensive simulations (24 hours) on the stages schedules. We look into the details of the completion time of the task from Figure 4.4(b). We placed the histogram obtained from the extensive simulations on top of the PERT distributions fitted on the histogram of limited simulations of 1000 runs from Figure 4.4(b). We consider a snapshot by zooming into the bottom portion to observe the outer bins of the histogram as shown in

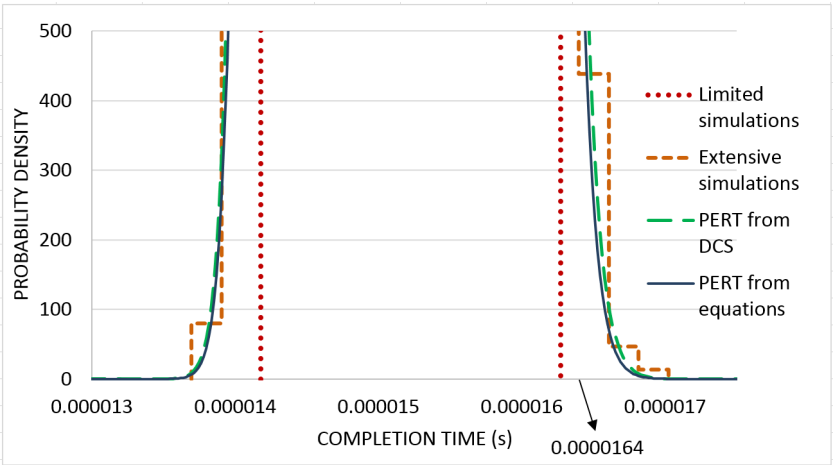


Figure 4.6: Comparison of PERT fit on limited and extensive simulations.

Figure 4.6. Note that the inner vertical lines are the outer edges of the outermost bins of the histogram from the limited simulations. Figure 4.6 makes it clear that the extensive simulations produced results outside of the bins of the histogram of the limited simulations.

The values of the metric M of the PERT using equations on the extensive simulations is 0.9905 and on the limited simulations is 0.9901. The M value of DCS is 0.9913 on the extensive simulations and 0.9902 on the extensive simulations. Both these results confirm that the PERT obtained from the limited simulations fits even better on the extensive simulations results. Hence, we observe that PERT is a good distribution to perform robustness analysis of a task, which cannot be done accurately with only limited simulations. We also observe that DCS again fits better on the extensive simulations than PERT equations.

The probability of deadline miss assuming, for the sake of the example, a deadline of $1.64 \cdot 10^{-5}s$ is 0 from the limited (1000) simulations, $0.69 \cdot 10^{-4}$ from the PERT equations and $0.98 \cdot 10^{-4}$ from DCS. On the other hand, the probability of deadline miss at $1.64 \cdot 10^{-5}s$ is approximated as $1.04 \cdot 10^{-4}$ from the extensive simulations. This shows that the PERT derived from the limited simulations gives reasonable estimates for events that only occurred in the extensive simulations. We also notice that DCS is more accurate in predicting the deadline miss probability of this task. However, the average number of tasks that miss deadlines in the extensive simulations is equal to 78.78 which is close to our prediction of 78.80 from PERT equations. Although DCS produces better curve fits on histograms than the PERT equations and more accurate deadline miss probability for this particular task, the PERT equations technique happens to give a more accurate schedule robustness result for the stages schedule. To understand this, we created

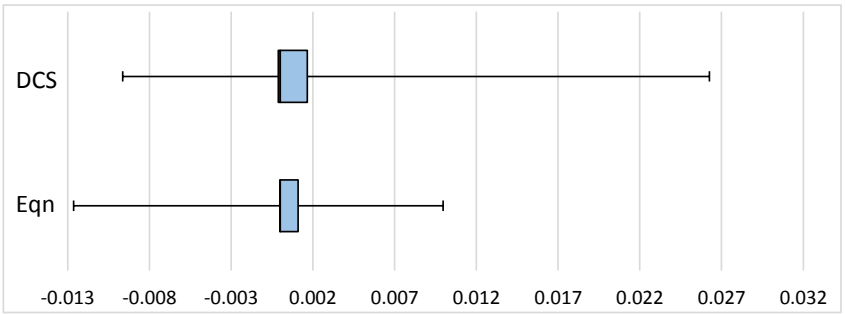


Figure 4.7: Box plot of deviation of the task deadline miss probabilities of the two curve fitting techniques with respect to extensive simulations.

box-plots of the deviations of the task deadline miss probabilities obtained using the two curve fitting techniques to the approximations from extensive simulations in Figure 4.7. We only include 164 tasks in the plot for whom at least one of the three techniques (robustness analysis with the two curve fitting techniques and extensive simulations) gives a deadline miss probability above $1 \cdot 10^{-5}$. This is to avoid the plot being dominated by the small probabilities of all the remaining tasks making it hard to visualize the deviations. We see in Figure 4.7 that the range of typical deviations of DCS, shown by the boxes, is bigger than for PERT equations. This explains why the overall schedule robustness is less accurate for DCS despite giving good curve fit. A reason for this could be the small loss of information that happens when abstracting the simulated completion time samples into histograms. However, both techniques overall give us quite accurate robustness results. From this we can conclude that our robustness analysis approach is sufficiently fast to be practically useful, whereas using extensive simulations to obtain statistically relevant results is not practically feasible.

4.5.3 Speed vs. accuracy: trade-off

Choosing one of the two curve-fitting approaches. The robustness analysis technique presented in this chapter is used within the robust scheduler in the next chapter. The speed of the robustness analysis then significantly contributes to the scheduler run-time. Hence, it is important that the run-time of the analysis is low. Although DCS performs better in curve fitting, it is a lot slower than PERT equations. Additionally, the difference in the accuracy of the results in quite small in comparison to the difference in run-times. Based on this, we make the choice of adopting the curve fitting using PERT equations to be used in the robustness analysis for the scheduler. The nested divide and conquer search technique is still

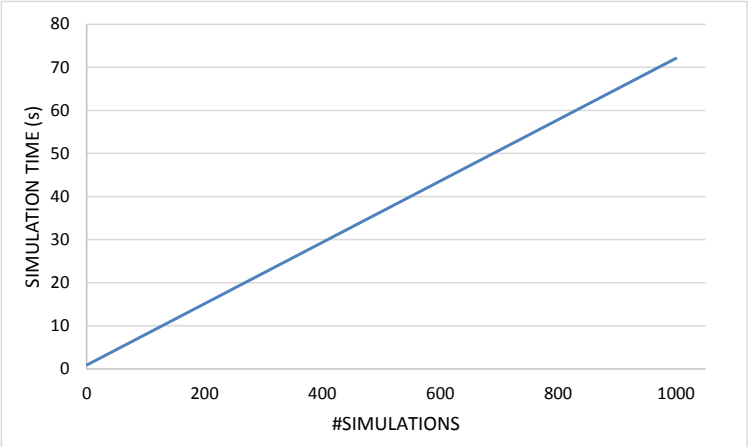


Figure 4.8: Plot of time taken to simulate different number of samples for the NXE stages schedule.

valuable as it can be generalized to other uni-modal distributions than PERT for other domains. The more the number of unknown parameters, the more involved is the nesting that is needed in the divide and conquer search though.

Number of simulations vs accuracy. By choosing the PERT equations over DCS, we gain significant time in the robustness analysis at a small cost of accuracy. Another time consuming part of the analysis is the simulations. For large graphs, we can save time by reducing the number of simulation runs. Figure 4.8 shows the time taken for the simulations increases linearly with the number of samples generated. However, reducing samples can affect the accuracy of the results. To study this impact, we calculate the expected numbers of deadline misses for different number of simulation runs lower than 1000 for the stages application and plot the results in Figure 4.9. On the x-axis are the number of simulation runs in the robustness analysis; on the y-axis are the corresponding expected values. We also plot a line corresponding to the average number of tasks that miss deadlines in the extensive simulations. We see that the accuracy of the results is low for few simulations, but it soon stabilizes towards the value obtained from extensive simulations and we can achieve nearly accurate results even with a few hundred simulations. The number of simulations thus provides a parameter to tune the accuracy vs. speed trade-off of the approach.

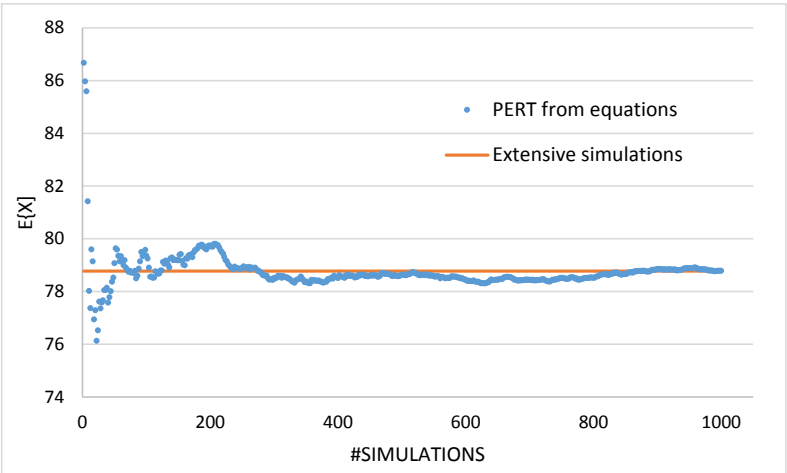


Figure 4.9: Plot of the expected number of deadline misses for different number of simulation runs in the robustness analysis for the NXE stages schedule.

4.6 Summary

In this chapter, we have presented an approach to perform robustness analysis of multiprocessor schedules. The overall approach addresses the complexity of performing max-plus operations on distributions by using a combined analytical and limited simulations-based approach. The metrics that we obtain quantify the robustness of tasks and schedules. We also present a new curve fitting metric to quantify how well a PERT distribution fits on a histogram. We present two ways of fitting a PERT distribution on the results of the limited simulations using analytically computed bounds, one using nested divide and conquer search and one using PERT equations. The PERT equations technique is much faster than the divide and conquer search while still being quite accurate. Hence, it is chosen to be used within the scheduler in the next chapter. The key contributions of this chapter are: 1) the overall robustness analysis technique that takes correlations due to dependencies naturally into account, and 2) the robustness metrics to quantify robustness of tasks and schedules. This robustness analysis technique is used in the next chapter to make scheduling decisions that steer the scheduler towards achieving more robust schedules.

Chapter 5

Iterative robust scheduling with fixed task binding

This chapter presents an iterative robust scheduling mechanism that uses the robustness analysis presented in Chapter 4 to produce robust schedules. The contents of this chapter is based on [5]. Given a DAG with fixed task binding, it begins with the schedule produced from the scheduler in Chapter 2 and iteratively improves its robustness. Robustness of a schedule is quantified using the metric from Chapter 4 based on the expected number of tasks that miss their deadlines in the schedule. This analysis is extended to compute a new metric that quantifies the task-level impact on the expected deadlines misses. This metric is used in an iterative highest robustness impact first heuristic to guide the stochastic list scheduler towards a robust schedule during the iterations. The scheduler repeatedly performs the extended robustness analysis followed by list scheduling with this new stochastic robustness heuristic to improve schedule robustness. The process stops either upon reaching a fixed number of iterations or if there are no changes in the schedules of successive iterations. The remainder of this chapter is organized as follows. Section 5.1 summarizes related work. Section 5.2 defines the problem by first introducing the preliminaries followed by the problem statement and the flow of the solution approach. Section 5.3 describes the extended robustness analysis and the new metric. Section 5.4 details on the scheduling heuristic and Section 5.5 elaborates on the iterative robust scheduler that uses the heuristic. Experiments are given in Section 5.6 and a summary of the chapter

is given in Section 5.7.

5.1 Related work

Task execution time variations that impact schedule robustness can arise due to multiple factors, resulting in different models of variation. In [10, 32], variations are modeled as external perturbations and their effects on system performance are used to study robustness. We model variations in task execution times which are internal to the system, due to the general purpose processing (cache misses, branch predictions, memory access delays). External perturbations do not play a role. Variations can be modeled using deterministic parameters such as min/max bounds and expected values, allowing for slack based robust scheduling techniques [18]. We use the richer model of probability density functions. Often, normal distributions are used [25, 29]. However in most practical cases, variations are not symmetric and are skewed in nature. We base our robustness analysis technique on realistic skewed bounded PERT distributions of execution times as explained in Chapter 4.

Robustness can be quantified using different metrics such as slack based metrics [29]. Stochastic metrics include more information. Deadline miss probabilities of tasks [83] is an example metric. This does not give a global measure of schedule robustness if there are multiple tasks with different deadlines. Global schedule robustness can be quantified using the probability that the makespan is within specific bounds [67]. This corresponds to all tasks having the same deadline. Our work considers task level deadlines which can differ within a graph.

Multiprocessor real-time scheduling with fixed execution times has been studied extensively, a survey of which is presented in [30]. In [29], robust scheduling using slack introduces temporal slack to each task such that a certain level of uncertainty can be absorbed without rescheduling. Here, normal distributions are used to describe machine breakdown behaviours and their parameters are used to calculate the amount of slack added per task. Task ordering is not changed. Introducing unnecessary slack can adversely effect meeting latency requirements. In our work, we alternatively determine stochastic task robustness and derive metrics that allow us to produce different task orderings that result in more robust schedules. A partitioning heuristic [33] takes execution time overruns into account and aims to assign tasks to resources such that the amount of allowable overrun per task is maximized. Its focus is on the assignment problem using fixed priority and deadline monotonic priority assignment. This chapter finds robust ordering of tasks for a given binding on resources and aims to maximize probabilities of meeting task deadlines.

Recent work on robust scheduling [55] looks at non-preemptive scheduling under task duration uncertainty. It presents a hybrid offline/online technique to meet hard real time guarantees with limited idle time insertion called Precedence Constraint Posting. Bounds and expected values are used instead of probability

distributions to deal with uncertainty. Tasks are assigned a common deadline corresponding to global real-time constraints. The scheduling aims to reduce makespan. In our context, each task has its own deadline and reducing makespan does not guarantee all tasks meeting their deadlines. As meeting latency requirements is our primary goal, we quantify robustness in terms of the expected number of task deadline misses and provide a scheduling heuristic to improve this. We use probability distributions for task execution times to analyze schedule robustness allowing for more accurate analysis in comparison to using bounds alone.

Other recent work on scheduling precedence constrained stochastic tasks on heterogenous cluster systems performs list scheduling based on stochastic bottom levels and stochastic dynamic levels [53]. It aims to reduce schedule makespan and does not address robust scheduling under task deadline constraints. The expected schedule makespan is measured by assuming that tasks have normal execution time distributions and using Clark's equations to estimate the max of distributions due to task synchronization [25]. Our work allows skewed task execution times distributions by adopting and refining the robustness analysis of Chapter 4. It combines analytical results with limited simulations. It overcomes the drawback of Clark's equations which are not accurate when performing max operations on distributions with large differences in their standard deviations [58]. Our analysis forms part of the iterative robust scheduling loop. Other alternatives are learning techniques such as metaheuristics [17] that explore the search space using heuristics to find good solutions. These can be quite time consuming depending on the number of solutions explored. We present an iterative list scheduler that can produce robust schedules in a small number of iterations.

Effective online scheduling approaches that deal with execution time uncertainty include reservation-based scheduling such as the Constant Bandwidth Server (CBS) introduced in [2]. Using a task model which comprises hard real time and soft real time tasks, they first schedule hard real time tasks based on latency requirements and consequently use reservation methods for soft real time tasks. These tasks are assigned dedicated processor bandwidth that can be reclaimed by other tasks on early completion. However in our task model, all tasks are soft real time and the scheduling aim is to reduce the number of possible deadline misses by making a robust static-order schedule. We consider scheduling that is performed during machine start-up time for ASML, or possibly on design time for other systems, whenever the application graph is available. The static order schedule thus produced cannot be changed online and tasks are run using a dispatcher in the same order. This is typical in throughput and latency driven applications that try to minimize any scheduling overhead during system run-time.

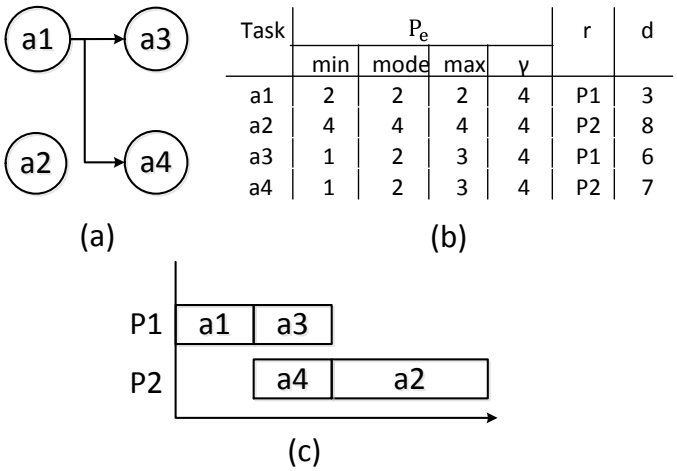


Figure 5.1: Example to illustrate list scheduling

5.2 Problem definition and solution overview

5.2.1 Preliminaries

The definition of tasks, dependencies and static-order schedules are the same as in Chapter 4. An addition is that tasks have a unique most-likely execution time, called the *nominal* execution time which is given by the mode or peak of the distribution (assuming single peaked distributions like PERT). We obtain execution time distributions by profiling and then use the chosen curve fitting technique using PERT equations from Chapter 4 to obtain modified PERT distributions. An alternative method to obtain execution time distributions is to use static probabilistic worst-case execution time analysis [40]. Like in Chapter 4, we assume that task execution times vary independently. Dealing with task execution time correlations is left for future work. The nominal start and completion times of a task are derived by using nominal execution times for all tasks in the graph. A schedule is *feasible* if all its tasks meet their deadlines under nominal execution times.

The iterative robust scheduling mechanism presented in this chapter is compared with the list scheduler with the EDDF heuristic from Chapter 2 that uses nominal execution times of tasks. To illustrate the flow of the approach in this chapter with an example, we consider the task graph in Figure 5.1(a). The attributes of the tasks are given in Figure 5.1(b). This is a simple case where the due-dates of task a_1, a_2, a_3 and a_4 are equal to their deadlines of 3, 8, 6 and 7

respectively. In the beginning, only a_1 and a_2 are enabled since they do not have any predecessors. Based on the earliest due-date first heuristic, a_1 is chosen to be scheduled. Thereafter, a_3 and a_4 become enabled along with a_2. Again based on due-dates, a_3 is scheduled followed by a_4 and then a_2 resulting in the schedule shown in Figure 5.1(c).

We use the robustness analysis technique of Chapter 4. We measure robustness of a static-order schedule S is terms of the expected value of the number of tasks that miss their deadlines in S using the following equation from Chapter 4.

$$E[X] = \sum_{a \in T} \mathbb{P}[c_a^S > d_a] \tag{5.1}$$

5.2.2 Problem statement and solution flow

The exact statement of the problem of this paper is as follows:

Problem Statement 4. *Given an application with tasks bound to a set of resources, find the static-order multiprocessor schedule with the lowest expected value of the number of tasks that miss deadlines.*

Producing a schedule with the maximum likelihood of meeting an optimization criterion under processing time uncertainty such as total flow time is proven to be NP-hard in [28]. Our problem is a generalization of this as we can connect all leaf tasks in the DAG to an additional task which has the flow time as its deadline. Hence, our problem is NP-hard and finding the optimal robust schedule is computationally intractable. We aim to design a stochastic robust scheduler that, starting from a certain schedule (produced using a non-stochastic scheduler in our case) uses an efficient scheduling heuristic to improve schedule robustness.

Our scheduling approach iterates between a scheduling step and a robustness analysis step. The scheduling step uses list scheduling with a heuristic to improve robustness. Our approach uses the deterministic scheduler from Chapter 2 that produces a starting schedule using nominal execution times of tasks. Given this schedule, our work aims to iteratively produce more robust schedules. The idea is to use the robustness analysis of the schedule obtained in an iteration to guide the scheduler towards a more robust schedule in the next iteration. Figure 5.2 illustrates the flow of the iterative robust scheduler. It begins with schedule S_0 obtained from the list scheduler with the EDDF heuristic. We perform robustness analysis of S_0, involving computation of the task completion time distributions, to compute a robustness *impact metric* per task. These metrics obtained from S_0 are used to refine the list scheduling towards robustness with an *iterative highest robustness impact first* (IHRIF) heuristic resulting in schedule S_1. This list scheduler with IHRIF heuristic differs from the EDDF list scheduler of the initial iteration only in the heuristic that determines the scheduling choices. The construction of the schedule, however, is done using nominal execution times of tasks for scheduling speed and simplicity's sake. This is because maintaining and

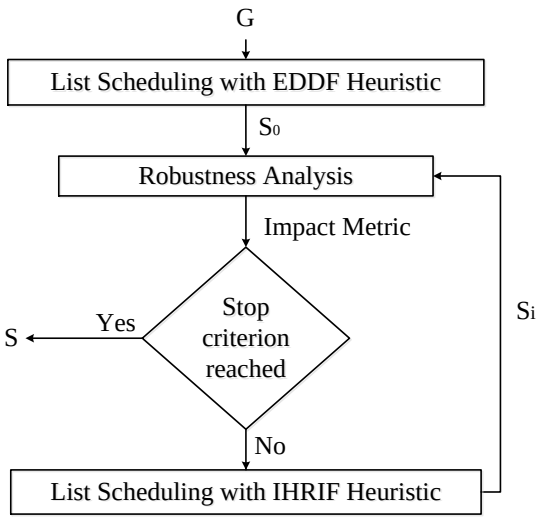


Figure 5.2: Flow of Iterative Robust Scheduler

updating start and completion time distributions during schedule construction is very expensive. In the next iteration, the impact metric computed from S_1 is used in the list scheduling heuristic to obtain S_2. This repeats until either the process converges with no schedule changes or until a fixed number of iterations.

To illustrate the flow of the scheduler, we reuse the example from Figure 5.1(a). The initial schedule S_0 is the one produced using list scheduling with the earliest due-date first heuristic and is given in Figure 5.1(c). We perform robustness analysis on this schedule. The expected value of deadline misses is 0.5 due to the deadline miss probability of a_2. The task completion time distributions, deadline miss probabilities and impact metrics of the tasks in S_0 are given in Figure 5.3(a). The computation of the impact metric is explained in Section 5.3.2. Tasks a_1, a_3 and a_4 have zero impact since the completion time distributions of their dependent tasks are within their own deadlines. Task a_2 has a positive impact since its deadline is in the middle of its completion time distribution. In the next iteration, list scheduling is performed using the highest impact first heuristic. As a result, a_2 gets priority and is scheduled first resulting in the schedule S_1 given in Figure 5.3(b) and the corresponding task details in Figure 5.3(c). The expected value of deadline misses of this new schedule obtained after performing robustness analysis is 0 implying that it has better robustness than the initial schedule.

The explanation of the iterative robust scheduler is divided into three sections. Section 5.3 elaborates on the refinement of the robustness analysis to compute the impact metric. Section 5.4 explains the IHRIF heuristic which uses the impact metric computed from the refined robustness analysis and Section 5.5 details on

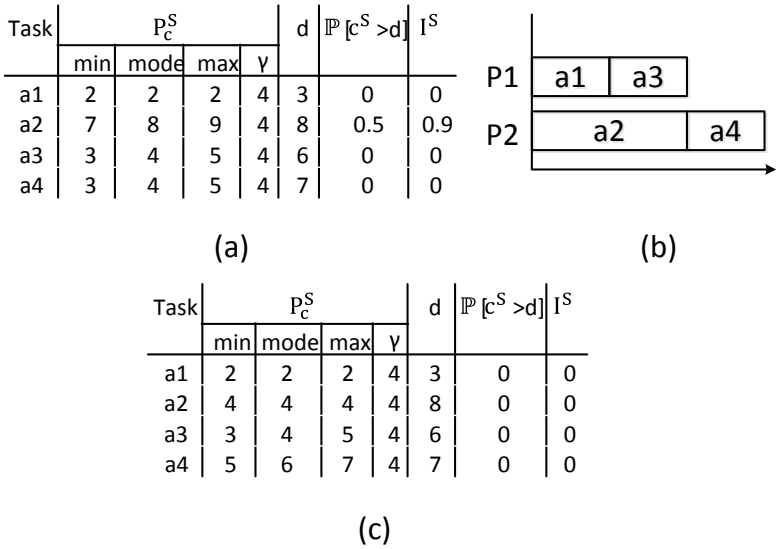


Figure 5.3: Example to illustrate the Iterative Robust Scheduling flow

the list scheduler with the IHRIF heuristic.

5.3 Refined robustness analysis: Impact metric

The robustness analysis of Chapter 4 uses task deadline miss probabilities to compute the schedule robustness using Equation 5.1. We refer to the schedule-level notion of robustness as *global robustness* and task-level notion of robustness as *local robustness*. Since list scheduling is performed on a per task basis, the global notion of robustness is insufficient to construct a robust schedule. For construction, we require a local notion of robustness that is consistent with and aids in the improvement of the global robustness. Hence, we refine the current robustness analysis with a new impact metric per task that provides this local notion of robustness.

5.3.1 Impact metric

Given a schedule S, this metric quantifies the impact of a task on the expected number of deadline misses in S. We use task deadline miss probabilities to extract the impact metric per task. The hypothesis is that schedule robustness can be improved by scheduling highest impact tasks first. The rationale behind the

heuristic is that delaying a task which has a high impact on the deadline misses of its future tasks will cause an increase in the expected deadline misses.

Definition 9. (Impact Metric) *The impact metric of a task 'a' at a given completion time c_a^S in a schedule S, denoted by $I_a^S : \mathbb{R}^{\geq 0}$, is the derivative of the expected value of number of task deadline misses in S over the delay in c_a.*

To estimate the impact metric of a task, we define the dependent set of a as follows.

Definition 10. (Dependent set) *Dependent set of task 'a', denoted by DS_a, is the set of tasks including a and all tasks that have a direct or indirect dependency from a in the DAG. These do not include dependencies due to scheduling order.*

Since the scheduling of task a influences the scheduling of the tasks in DS_a, we need to compute the expected value of the number of tasks that miss deadlines in DS_a to compute I_a^S. Let random variable X_a represent the number of tasks from DS_a that miss deadlines in S. The expected value of X_a is given by

$$E[X_a] = \sum_{b \in DS_a} \mathbb{P}[c_b^S > d_b] \tag{5.2}$$

The value of $E[X_a]$ as a function of the completion time of a is denoted by the function $v_{E[X_a]} : \mathbb{D} \to \mathbb{R}^{\geq 0}$. The derivative of $v_{E[X_a]}$ gives the impact metric representing the impact on $E[X_a]$ caused by delaying c_a^S by a small amount of time. To express the shifting of a probability density function by an amount of time $\tau \in \mathbb{R}^{\geq 0}$, we define ${}_\tau d : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ such that ${}_\tau d(t) = d(t - \tau)$ for all $t \in \mathbb{R}$. Using this, ${}_\tau c_a^S$ represents the random variable for the completion time of a after the shift of τ and ${}_\tau P_{c_a}^S$ represents the corresponding probability density function. This in turn causes an increase of 'at most' τ in the completion times of tasks in DS_a. From Equation 5.2, we deduce that this may cause the expected value of X_a to increase. Hence, the impact metric of a in S is captured as follows.

$$I_a^S = \lim_{\tau \to 0} \frac{v_{E[X_a]}({}_\tau c_a^S) - v_{E[X_a]}(c_a^S)}{\tau} = \frac{dv_{E[X_a]}({}_\tau c_a^S)}{d\tau} \tag{5.3}$$

In the next subsection we explain the computation of this derivative.

5.3.2 Impact metric computation

In schedule S, any change in $E[X_a]$ due to delaying a is only caused by tasks in DS_a that experience an increase in their deadline miss probabilities due to the delay. To compute the derivative in Equation 5.3, we consider the delay to be infinitesimal. This in turn can cause only an infinitesimal shift in the completion times of tasks in DS_a. The increase in the deadline miss probability of a task

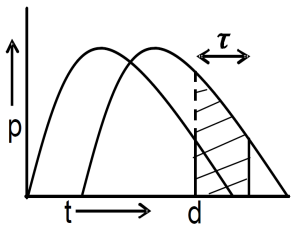


Figure 5.4: Increase in deadline miss probability due to delaying a task

$b \in DS_a$ for an infinitesimal shift in c_b^S is equal to the probability density at its deadline:

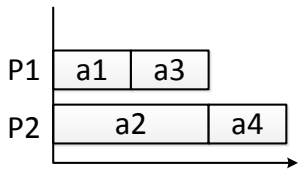
$$\lim_{\tau \to 0} (\mathbb{P}[_\tau c_b^S > d_b] - \mathbb{P}[c_b^S > d_b]) = P_{c_b}^S(d_b) \tag{5.4}$$

Here, $\mathbb{P}[c_b^S > d_b]$ is the deadline miss probability of b before the shift and $\mathbb{P}[_\tau c_b^S > d_b]$ gives the deadline miss probability after shifting by τ. The probability density at its deadline d_b is given by $P_{c_b}^S(d_b)$. How we arrive at this is illustrated in Figure 5.4 that shows a task's completion time distribution shifted by τ. This causes an increase in its deadline miss probability given by the shaded portion. For an infinitesimal increase $\tau \to 0$, this area collapses into a straight line of length equal to the probability density at the task deadline d.

Another important aspect of the computation is that shifting the completion time of a by τ will increase the deadline miss probability of task $b \in DS_a$ only if the slack between a and b is smaller than τ. This slack in S, denoted by sl_{ab}^S, is the difference between the completion time of a and the start time of b taking dependent tasks in between into account. Taking the limit $\tau \to 0$ implies that task b will be affected iff $sl_{ab}^S = 0$. Note that sl_{ab}^S can never be smaller than 0. The probability of sl_{ab}^S being 0, denoted by $\mathbb{P}[sl_{ab}^S = 0]$, can be approximated by counting the number of times the slack equals zero in the simulations used in the robustness analysis. These probabilities, starting from immediate predecessors of leaf nodes, are propagated backwards through the schedule to get the probabilities also for tasks which do not have a direct dependency between them. Also, the probability of zero slack between a task and itself, given by $\mathbb{P}[sl_{aa}^S = 0]$, is 1. Using this, we compute the impact metric of a as follows.

$$I_a^S = \sum_{b \in DS_a} \mathbb{P}[sl_{ab}^S = 0] \cdot P_{c_b}^S(d_b) \tag{5.5}$$

Here, we see that the impact metric of a task is based heavily on its successors and can be computed only after the successors have been scheduled. Hence, computing I_a^S requires the complete schedule of all tasks in DS_a. Similarly any change in the



(a)

Task	P_e				r	P_c^S				d	I^S
	min	mode	max	γ		min	mode	max	γ		
a1	2	2	2	4	P1	2	2	2	4	3	0.9
a2	4	4	4	4	P2	4	4	4	4	5	0
a3	1	2	3	4	P1	3	4	5	4	4	0.9
a4	1	2	3	4	P2	5	6	7	4	6	0.9

(b)

Figure 5.5: Example to illustrate the Impact Metric Computation

scheduling of tasks in DS_a might require I_a^S to be recomputed. Therefore, the robust scheduling approach presented in this work uses an iterative mechanism and cannot be performed efficiently in one shot. We compute the impact metrics of tasks in a schedule generated in the previous iteration and use it for scheduling in the next iteration. The idea behind this approach is to progress towards a more robust schedule by performing robustness based scheduling decisions in an iteration, for high impact tasks detected in the schedule of the previous iteration.

To illustrate the impact metric computation we reuse the example schedule in Figure 5.3(b) but with tighter deadlines to get non-zero impacts. The execution time and completion time distributions, deadlines, and impacts of the tasks are in Figure 5.5(b). Impacts of a_2, a_3 and a_4 having no successors are equal to the probability densities of their completion distributions at their respective deadlines. The probability of zero slack between a_1 and itself is 1. The probability of zero slack between a_1 and a_3 is also 1 (a_3 always immediately follows a_1). The probability of zero slack between a_1 and a_4 is 0 (the start time of a_4 depends on the completion time of a_2 which is always greater than that of a_1). Hence, the impact of a_1 is computed as follows.

$$\begin{aligned} I_{a_1}^S &= \mathbb{P}[sl_{a_1a_1}^S = 0] \cdot P_{c_{a_1}}^S(d_{a_1}) + \mathbb{P}[sl_{a_1a_3}^S = 0] \cdot P_{c_{a_3}}^S(d_{a_3}) \\ &\quad + \mathbb{P}[sl_{a_1a_4}^S = 0] \cdot P_{c_{a_4}}^S(d_{a_4}) \\ &= 1 \cdot 0 + 1 \cdot 0.9 + 0 \cdot 0.9 = 0.9 \end{aligned}$$

Note that different deadlines can result in different impact values and different scheduling choices for the same graph. In the next subsection we elaborate on the IHRIF heuristic which uses the impact metric to guide the scheduling decisions.

5.4 Iterative highest robustness impact first heuristic

The iterative approach uses a list scheduler which chooses tasks from the list of enabled tasks using the IHRIF heuristic. The various components of the heuristic are as follows.

1) *Impact History*: During the robustness analysis of S, we only obtain the impact of a at its current completion time in S. At different decision points during scheduling, the completion time of a can be different from the ones observed in prior schedules. We need to be able to estimate impact values at all possible completion times. To do so, we maintain a history of all the impact values observed in prior iterations along with their corresponding nominal completion times. We use nominal completion times mainly to simplify the approach by removing the complexity of combining density functions in the impact history. With more iterations, we obtain more values and can approximate the impact metrics of tasks with changing completion times.

Definition 11. (Impact History) *The impact history of a task 'a' is a function which maps an iteration i to a tuple of its nominal completion time and the corresponding impact metric at that iteration. It is denoted by the mapping $IH_a : \mathbb{N} \rightarrow (\mathbb{R}^{\geq 0}, \mathbb{R}^{\geq 0})$. The functions $left(IH_a(i))$ and $right(IH_a(i))$ give the nominal completion time and the impact metric of a at the i^{th} iteration.*

Using this function during scheduling, we need to deduce an *impact estimate* of a at the potential nominal completion time of a if it were to be scheduled next. This impact estimate enables us to approximate the impact metrics at completion times not observed in schedules of prior iterations. At each scheduling step we need to compute impact estimates for all enabled tasks. The IHRIF heuristic then ranks enabled tasks based on the highest impact estimate. At each scheduling step, the task with the highest impact estimate is scheduled.

Definition 12. (Impact Estimate) *The impact estimate of a task 'a', used during scheduling, is a function which maps a potential nominal completion time of a to the estimated impact metric at that time using the current impact history. It is denoted by the function $IE_a : \mathbb{R}^{\geq 0} \rightarrow \mathbb{R}^{\geq 0}$.*

We can derive impact estimates at any point in time from the points in the impact history. Since the impact computation performed during robustness analysis of a schedule is a random process, we can have multiple impact metric values for the same nominal completion time. These values can arise if we compute the impact of a task the position of which has not changed in multiple iterations. This could also happen if two different schedules result in the same nominal completion time for the task. To smoothen out the effects of these multiple values, we use a weighted distance average computation to compute impact estimate from the impact history. If CH_a is the set of all nominal completion times of a in the

impact history and K the number of iterations so far, then the following equation is used to compute IE_a at time t from its impact history.

$$IE_a(t) =$$

$$\begin{cases} \text{avg}\{right(IH_a(k)) | left(IH_a(k)) = min(CH_a)\} & t < min(CH_a) \\ \text{avg}\{right(IH_a(k)) | left(IH_a(k)) = max(CH_a)\} & t > max(CH_a) \\ \text{avg}\{right(IH_a(k)) | left(IH_a(k)) = t\} & t \in CH_a \\ \frac{\sum_{k \in K} \frac{\Delta_k}{d_k} \cdot right(IH_a(k))}{\sum_{k \in K} \frac{\Delta_k}{d_k}} & \text{otherwise} \end{cases}$$

where $d_k = |left(IH_a(k) - t|$ is the distance from t to $left(IH_a(k))$ and $\Delta_k = \frac{left(IH_a(k+1)) - left(IH_a(k-1))}{2}$ denotes the range of values covered by the k^{th} entry. Multiplying the weight by Δ_k ensures that having multiple completion time entries close to each other does not bias the impact estimate in that region.

2) *Scheduling history (fallback choices)*: The impact history allows us to improve the estimation of the impact metrics used in the scheduling heuristic. Aside from this, we also want to improve the scheduling decisions by maintaining the *scheduling history*. This involves keeping the scheduling decisions of the best prior schedule in order to steer the search process of the heuristic towards global robustness improvement. Scheduling decisions are defined below.

Definition 13. (Scheduling Decisions) *Scheduling decisions of a scheduler, denoted by the function $SD : T \to \mathbb{N}$, is a function which maps a task to the step number at which the scheduler schedules the task.*

Maintaining scheduling history implies that we base our fallback choices on the *scheduling decisions* made in the most globally robust schedule observed so far in all prior iterations. This is the schedule with the lowest expected number of deadline misses. Fallback choices occur when all tasks in the enabled set have the same impact estimate. The reason for basing the fallback choices on the scheduling decisions of this schedule is to utilize the knowledge that these decisions have previously led to a globally more robust schedule.

3) *Impact Threshold*: An additional aspect of the heuristic is to limit the changes in the schedule to high impact tasks of each iteration. The rationale is based on the fact that the impact history is derived from schedules observed in prior iterations. If drastic changes are made early on during scheduling for low impact tasks, the schedule being formed becomes too different. This renders the impact estimate inaccurate for the subsequent high impact tasks which become visible to the scheduler only at the later scheduling steps. Therefore, we introduce an *impact threshold*, denoted by it, such that only impact values above a certain threshold are considered during scheduling. For tasks with impact values below

the threshold, the scheduler uses the scheduling history explained earlier. This threshold is computed as a certain percentage, called *impact threshold percentage itp*, of the highest impact metric observed in the last iteration. By doing this the low impact tasks in a current iteration become visible above the threshold only in the later iterations, when the scheduler has already dealt with the current high impact tasks.

5.5 Iterative list scheduling with IHRIF heuristic

Algorithm 9 gives the overall iterative robust scheduler. N_I is the number of allowed iterations, SD_i the i^{th} iteration scheduling decisions, ListScheduler$_{\text{EDDF}}$() the list scheduler with EDDF heuristic, ListScheduler$_{\text{IHRIF}}$() the list scheduler with IHRIF heuristic, and RobustnessAnalysis() is the robustness analysis step. Line 1 produces an initial schedule using ListScheduler$_{\text{EDDF}}$() and Line 3 performs RobustnessAnalysis(). From the impact metrics obtained, impact threshold is computed in Line 4. We iteratively perform ListScheduler$_{\text{IHRIF}}$() and RobustnessAnalysis() in the loop between Lines 6-14 until stop criterion of Line 14. Line 9 recomputes impact threshold for the next iteration. If the schedule produced is better than previous best, Line 10-12 updates the best schedule and scheduling decisions. Line 15 returns the best schedule. Note that by construction the best result S_{Best} cannot be worse than the EDDF schedule.

For complexity analysis of the iterative algorithm, we consider its 3 major components: (1) ListScheduler$_{\text{EDDF}}$(), (2) ListScheduler$_{\text{IHRIF}}$() and (3) RobustnessAnalysis(). ListScheduler$_{\text{EDDF}}$() includes topological sorting of tasks and computation of due-dates. Topological sorting is linear in the number of tasks and dependencies. Due-date computation uses an efficient linear approach requiring one backward traversal of the graph. ListScheduler$_{\text{IHRIF}}$() differs only in the scheduling heuristic involving the computation of the impact estimate from the impact history, whose entries depends on the number of iterations. Since we need to repeat the computation for every task in the enabled set along with obtaining their potential completion times, this heuristic has a quadratic complexity of $O(N_I \cdot (|T| + |D|)^2)$. RobustnessAnalysis() obtains completion time distributions of tasks by linear max-plus computation of completion time bounds and a limited number of simulations, N_{Sim}. This requires the traversal of the schedule N_{Sim} times and the complexity is $O(N_{Sim} \cdot (|T| + |D|))$. There is a trade-off here between the number of simulations and accuracy of the analysis which can be exploited to reduce the scheduler run-times. RobustnessAnalysis() additionally performs the impact metric computation per task as a product of two factors in Equation 5.5. Zero slack probability can be computed in one backward traversal through the graph. The second factor, probability density at the deadlines, takes constant time independent of graph size. Overall, since the impact metric of each task needs the sum of values obtained from all dependent tasks, this computa-

Algorithm 9: IterativeRobustScheduler()

Input : $G := (T, D), R, N_I, thp$
Output: S

$[S_0, SD_0] :=$ ListScheduler$_{\text{EDDF}}(G, R)$;
$IH = \phi$;
$[E[X_0], IH] :=$ RobustnessAnalysis(S_0, IH);
$it := \frac{itp}{100} \cdot \max_{a \in T}$ *right*$(IH_a(0))$;
$S_{Best} := S_0$, $SD_{Best} := SD_0$, $i := 0$;
repeat
  $[S_i, SD_i] :=$ ListScheduler$_{\text{IHRIF}}(G, R, IH, it, SD_{Best})$;
  $[E[X_i], IH] :=$ RobustnessAnalysis(S_i);
  $it := \frac{itp}{100} \cdot \max_{a \in T}$ *right*$(IH_a(i))$;
  if $E[X_i] < E[X_{i-1}]$ **then**
    $S_{Best} := S_i$, $SD_{Best} := SD_i$;
  end
  i:=i+1;
until $i \leq N_I$ *or* $S_i = S_{i-1}$;
return S_{Best};

tion is quadratic in the number of tasks and dependencies. Lastly computation of expected deadline misses involves obtaining the deadline miss probabilities of constituent tasks and summing them up. The computation of these cumulative probabilities takes constant time independent of graph size.

5.6 Experimental results

In this section, we evaluate the iterative robust scheduler on real world applications and multiple synthetic test sets.

5.6.1 Real world applications

Lithography Machines. To test the scheduler on the wafer scanners we chose the NXT stages application similar to Chapter 2 with fixed binding on single-core processors. It has 4301 control tasks and 4095 dependencies, with a degree ranging from 0 to 22, running with a frequency of 10kHz on a platform of 11 general purpose single-core processors. Task execution time distributions are estimated from measurements made on the machine. Critical actuator tasks are assigned 30% and the remaining tasks are assigned 70% of the processor budget as deadline. The expected number of tasks that miss deadlines in the EDDF schedule is around 458. A threshold percentage of 10% and 10 iterations gives a schedule with

an expected value of deadline misses of around 445 which is approximately 3% robustness improvement. The scheduler runtime was 8.55 minutes.

Video Conferencing Applications. With current advancements in video streaming, video conferencing on applications like Skype has become an integral part of daily life. Video and audio decoders are software components that convert a compressed video or audio into raw uncompressed form. The rate at which the decoders decompress frames has a direct impact on the quality of the video conference experience. The end to end delay can be seen as a latency requirement on the end task of the decoders. Missing these latency requirements results in jitter and failures experienced during the conference calls. We tested the robust scheduler on an application consisting of two audio and two video decoders. The graphs and the distributions are taken from documented Scenario Aware Dataflow Graphs (SADF) of video and audio decoders [74]. The graph of the four SADFs combined consists of 1982 tasks, each performing a software function, mapped onto a platform consisting on three general purpose processors. A common deadline is assigned to the end tasks of all the four decoders. The expected value of deadline misses from the EDDF schedule is 0.53, consisting of the end tasks of the two video decoders having a deadline miss probability of 0.24 and 0.29 respectively. Starting from this schedule, the robust scheduler produces a schedule with an expected value of deadline misses equal to 0.11 using a threshold of 10% and 10 iterations. In this schedule, the end tasks of the video decoders have a deadline miss probability of 0.05 and 0.06 respectively. This is a significant robustness improvement of 79% experienced during the conference calls. The scheduler run-time was 4.25 minutes with 100 simulations in the robustness analysis.

A variant of the above application includes three video decoders with 603 tasks on a platform consisting of two homogenous general purpose processors. We assign fewer resources to utilize the parallelism in the application. Tight deadlines result in an expected value of deadline misses of 2.39 in the EDDF schedule. The robust scheduler yields a schedule with an expected value of 2.16 with a threshold of 10% and 10 iterations which is around 10% improvement in robustness. The scheduler ran for 1.33 minutes.

5.6.2 Synthetic test cases

For further validation, we use multiple test sets consisting of 1000 DAGs generated using the random graph generator tool of SDF3 [72]. In the first test set, each graph consists of 100 tasks and each task has a degree ranging from 1 to 5 with an average of 2 and variance of 1. These tasks are bound to two to five resources such that highly interconnected sub-branches are mapped on the same resources to the maximum possible extent. This is to mimic the binding approach used in control applications wherein tasks in highly interconnected control loops are mapped on the same resources in order to minimize communication overhead.

Figure 5.6(a) compares the results obtained using the EDDF and the IHRIF scheduler. The horizontal axis represents the iterations in increasing order. The

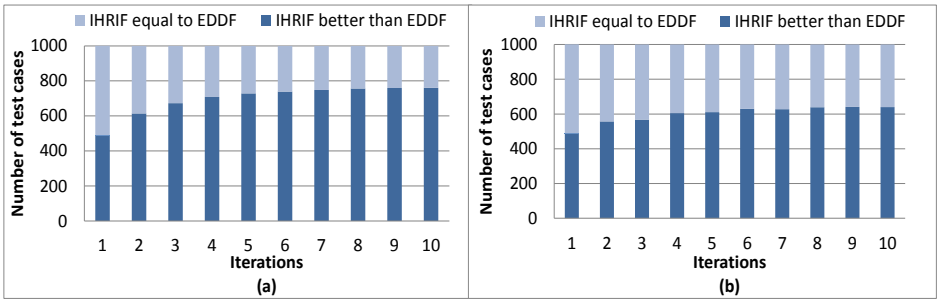


Figure 5.6: (a) Aggregated, and (b) per iteration schedule robustness improvements of IHRIF over EDDF

number of allowed iterations was chosen to be reasonably large at 10. The bottom portion of each vertical bar gives the number of graphs from 1000 that improved in robustness. The top portion gives the ones that showed no improvement. No improvement arises when IHRIF does not perform better than EDDF. In some of these cases the EDDF schedules may already be optimally robust. We see an aggregated result of 76% more robust schedules at the end of 10 iterations. It can be shown that this is statistically a very significant result.

Since 5.6(a) gives aggregated results (taking the best of all schedules upto each iteration). From this, we cannot assess whether the individual iterations also produce significantly better results. In other words, we would also like to see if with more iterations, each iteration of the scheduler by itself also, on average, produces more robust schedules. From this we can deduce whether the schedules converge over iterations. Figure 5.6(b) compares each iteration of IHRIF individually with EDDF. Also per iteration, the number of more robust schedules gets significantly better in Figure 5.6(b). This validates our hypothesis that maintaining history in the iterative approach allows the scheduler to gather sufficient information over iterations to produce more robust schedules.

Figure 5.7 compares the expected number of deadline misses in schedules obtained from EDDF, the X-axis, and from IHRIF, the Y-axis. Points below the line correspond to cases where IHRIF improves over EDDF. The test set consists of a range of problems with low and high expected number of deadline misses from the EDDF schedules; we see improvement in most cases. The absolute improvement ranges from 0 to 6 tasks. The relative improvement compared to EDDF schedules ranges from 0-100% with an average of 12%, a median of 5% and a standard deviation of 18% (which is high due to some exceptional very high robustness improvement cases), see Figure 5.8. There is no tractable algorithm to find optimally robust schedules. Exhaustive state space search does not scale for DAGs with more than a few tasks. Hence, we cannot compare to optimal schedules. To test whether the approach is sensitive towards the binding mechanism used and

to different task graph sizes, we also validated the approach on test sets with a first fit binding approach and on sets of larger task graph sizes with 500 and 4500 tasks. The results are consistent with the reported ones. The scheduler run-time for graphs of sizes 100 and 500 is tens of seconds per graph, using 1000 simulations in the robustness analysis. For larger graphs of size 4500, we use 100 simulations in robustness analysis and obtain run-times in the order of minutes.

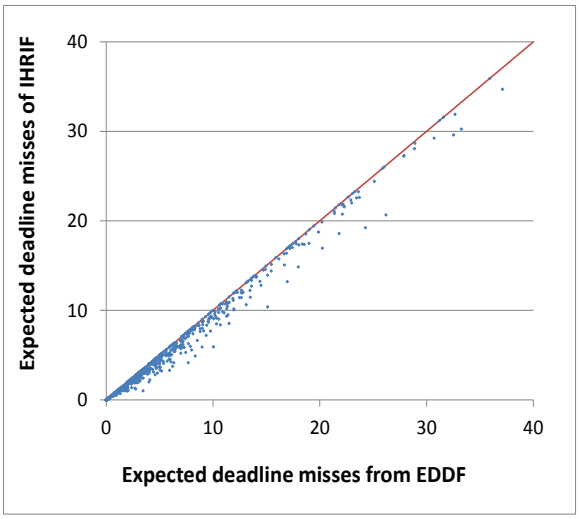


Figure 5.7: Comparison of the expected values of number of tasks that miss deadlines obtained from EDDF and IHRIF

5.7 Summary

This chapter presents an iterative robust scheduler. It begins with a schedule produced using nominal execution times and iteratively improves its robustness using a refined version of the robustness analysis from Chapter 4. The key contributions of this chapter are: 1) an overall iterative robust scheduling approach that improves the robustness of the starting schedule, 2) a new task impact metric that helps predict upfront the impact of delaying a task on the schedule robustness and 3) an IHRIF list scheduling heuristic based on the impact metric used by the scheduler to make scheduling decisions. The end result of the approach is a schedule with a lower expected number of deadline misses and thus a higher robustness. The iterative approach is necessary to allow estimating the impact of a task on robustness of a schedule, a piece of information that depends on the complete schedule that is not yet known during scheduling. The scheduler has

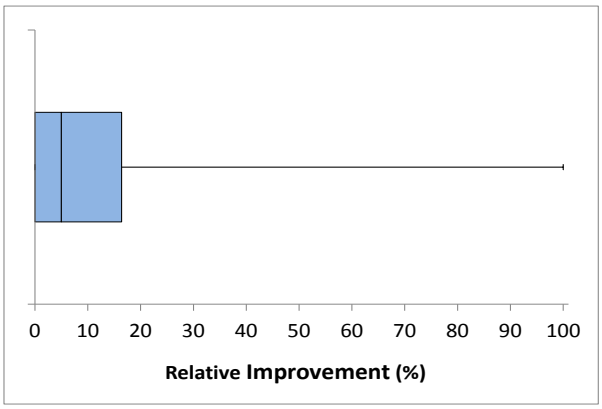


Figure 5.8: BoxPlot of relative improvement of IHRIF over EDDF

been validated on real world applications and has shown to deliver robustness improvements between 3% and 79%. It has also been tested on 1000 synthetic graphs and has shown improvements in 76% of the schedules with an average robustness improvement of 12%.

Chapter 6

Conclusions and future work

Many technological advances that are seen today have been possible due to the rapid growth and development of the semiconductor industry that has continuously reduced the feature size of a chip to push Moore's law. One of the primary reasons for this reduction is the increasing performance of lithography machines that are responsible for exposing patterns of electronic circuitry onto silicon chips. This performance increase comes at the cost of increasing complexity of the machines and the need for high performance control systems to ensure the smooth functioning of their components. The low IO delays and the high frequencies with which these control systems execute translate into very strict latency requirements on their control applications. To achieve the desired performance, these control applications are to be bound and scheduled on tens of general purpose multiprocessor platforms such that all latency requirements are met. The static-order schedules produced must be robust against execution time variations. Owing to the fact that several configuration parameters are not known beforehand, schedulers must run during the minimal machine initialization time. Therefore, they must be fast and scalable to the large control applications. These challenges are combined into the problem statement of this thesis to achieve fast and scalable robust communication aware binding and scheduling of DAGs on multiprocessor platforms, while meeting all latency requirements. We have broken down the problem statement into its primary aspects to develop solutions incrementally.

6.1 Conclusions

We addressed the various aspects of the problem statement of this thesis in four main contributions. In the first contribution, we tackle the first challenge of the thesis by presenting a fast and scalable multiprocessor scheduler for DAGs with fixed binding on multiprocessor platforms that aims to meet latency requirements. We compute a due-date metric for tasks with deadlines by exploiting the binding information of the application. We present a list scheduler that uses the earliest due-date first heuristic to increase the odds of finding feasible schedules in one attempt. It has been shown to compute feasible schedules of very large task graphs within little time. The heuristic has been shown to outperform the classical earliest CALAP first heuristic in producing feasible schedules for ASML control applications as well as for 1000 large synthetic task graphs. The approach has been incorporated in all the latest ASML lithography scanners.

Our second contribution is a communication aware binding method that deals with the second thesis challenge. It is a three-step binding algorithm, named CMA, that computes the binding of tasks on the resources of a shared memory multiprocessor platform. In three steps we capture the parallelism in the application, the latency requirements, the shared memory communication mechanism and the platform size limitations to derive a binding that results in feasible schedules with low makespan. In the first step we apply a deadline aware shared memory extension of the DSC algorithm that produces a clustered graph assuming unlimited resources. Since the number of clusters produced can be larger than the number of resources, we next perform a merging step to combine clusters based on dependencies while constraining them to remain smaller than a threshold. Finally the clusters are allocated to the available processors while balancing the workload. The three-step approach of CMA is shown to be particularly beneficial when dealing with the inaccuracy in the computation of synchronization timings for shared memory communication prior to scheduling. The binding algorithm has been validated in combination with the scheduler from Chapter 2 on control applications of ASML in which it is shown to produce lower makespans than the state of the art BDSC algorithm. It is also shown to outperform BDSC in a significantly high number of test cases of other well-known parallel problems. The binding algorithm has been adopted for integration by ASML.

The third contribution of this thesis is the first step towards the robust scheduling challenge wherein we develop an approach to perform robustness analysis of static-order multiprocessor schedules. We define metrics to quantify the robustness of tasks and schedules. Robustness of a task is defined as the probability of meeting its deadlines. Robustness of a schedule is defined in terms of the expected value of the number of tasks missing deadlines in the schedule. The overall approach addresses the complexity of performing max-plus operations on distributions by using an approach that effectively combines a best-case and worst-case analysis with limited simulations. We present two curve fitting techniques to de-

rive the distributions needed for the analysis. We select one based on its speed and accuracy to be used in the fourth thesis contribution. The approaches have been tested for scalability on wafer scanner control applications.

Lastly, we present an iterative robust scheduler that, starting from the schedule produced by the list scheduler with the earliest due-date first heuristic from Chapter 2, iteratively improves the schedule robustness. It uses the robustness analysis technique of Chapter 4 after refining it to produce a new task level impact metric that estimates the impact of delaying a task on schedule robustness. This new metric is used by the list scheduler that employs a new robustness heuristic called 'iterative highest robustness impact first'. The approach reduces the expected number of deadlines misses in a schedule thereby improving schedule robustness. The iterative mechanism is needed to extract sufficiently precise robustness information without excessive lookahead at each scheduling step. The scheduler has been validated on wafer scanner applications, video conferencing applications and 1000 synthetic graphs where it has shown significant robustness improvements.

6.2 Future work

In this thesis, we have addressed all the separate aspects of the problem statement except for the last challenge of developing one integrated framework that can deal with all of them together. The open points form directions for future work and are listed below.

6.2.1 Fast and scalable communication aware robust binding and scheduling

The binding technique presented in Chapter 3 does not consider execution time variations. A direct way to address these would be to incorporate the variations by computing stochastic distributions of the task attributes such as top levels and due-dates. With these distributions, we can obtain slack probabilities that can be used in the priorities for the clustering algorithm. The binding so obtained should then be fed into the robust scheduler from Chapter 5 after it is extended to also account for the shared memory communication overhead.

In addition to defining how to functionally combine all the aspects of the problem statement, another important issue is the impact on the scheduling time and scalability. The current iterative scheduler without communication aware binding is already a lot slower than the 'one shot' EDDF scheduler. The multi-step binding algorithm and the transitive reductions during scheduling are also computationally expensive and together they will not meet the scalability and speed requirements. Effective ways of approximating information in a meaningful manner will therefore be needed to speed up the scheduling process. However the

approach will still be quite useful in cases where scheduling time is not constrained, such as for offline scheduling or for scheduling of relatively small sized applications.

6.2.2 Robustness analysis and scheduling under communication contention

Throughout the thesis, we made the assumption that communication time between processors through the network and the read-write costs to the memory are accounted for in the execution times of tasks. However, communication resource contention is an important factor that causes variations in the communication timings and the subsequent cumulative execution times of tasks that follow them in the graph. Hence, these assumptions need to be relaxed to obtain robustness estimates that are much closer to the observations on the machines. Variations due to contention arise mainly because the communication through the network and memory hierarchy is no longer statically ordered. It depends on the arbitration policies of the network interconnect in case of inter-processor communication and on the shared memory hierarchy in case of communication between processor cores. In the interconnects at ASML, a first-come-first-served (FCFS) arbitration policy is used.

The robustness analysis presented in this thesis works only for static-order schedules. To extend it, we first need a means to compute the min-max completion time bounds of tasks taking also communication contention into account. An initial step in this direction, made in [36], presents a fix-point based interval analysis technique to compute the task completion time bounds for a DAG mapped on a set of interconnected resources that employ FCFS arbitration. Once we have established the min-max bounds, we need simulated samples for the distribution of the probability mass. The simulation models will need to mimic the entire communication framework with the arbitration policies so that we can obtain completion time samples in the presence of communication contention. Thereafter, we can perform curve fitting on the histograms of the simulated samples together with the interval analysis bounds to obtain completion time distributions and measure robustness. When integrating this analysis with robust scheduling, introducing a fix-point analysis at each scheduling iteration together with the multiple simulations will no longer be scalable to be applicable for ASML. However, as mentioned earlier the approach is still worthwhile for small sized applications or offline scheduling problems.

6.2.3 Robustness analysis under execution time correlations

In this thesis, we have ignored correlations between the execution times of tasks. In order to take these correlations into account during robustness analysis, we need a means to first extract the coefficients that define the correlation between the different execution time variables. Once this is known, we need to employ

more involved sampling techniques that can draw samples from conditional distributions of the random variables taking their correlations into account.

6.2.4 Robustness analysis for other application domains

The robustness analysis presented in this thesis is also applicable to other domains such as the study of the impact of delays in timetabled systems such as the railways. It is often seen in day to day commuting that a delay in the arrival of one train to a particular station can significantly impact the schedules of the others trains passing through that station. In this domain, the scheduling of trains needs to be robust against the delays caused by system failures or weather conditions. The travelling times of trains can be expressed in terms of execution time distributions and deadlines would then correspond to their arrival times at stations. To focus on the impact of delays alone, the robustness analysis can be extended to further improve the accuracy of the PERT distributions at their tails. This can be achieved by using importance sampling techniques that allow us to generate samples for rare events and perform even better fitting of distributions near the tails.

6.2.5 Multi-rate scheduling: Data flow models

The applications studied in this thesis are all typically running at the same sample frequencies. In ASML, some control applications are more critical than others and the designers often decide to execute them at relatively higher sample frequencies. The current scheduling framework cannot deal with applications running at multiple frequencies on the same processors. For example the POB application, considered in Chapter 3, consists of two independent parts running at different sample frequencies that are executed on different processors. With the introduction of multi-rate control, we will need to evolve from our DAGs to more expressive formalisms such as Synchronous Data Flow (SDF) [52] graphs that use token rates to express the relative execution rates of tasks. The computations at each scheduling step will become more involved to also keep track of these rates and schedule tasks accordingly.

Bibliography

[1] ASML. `http://www.asml.com`. Accessed: 08/03/2016.

[2] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 4–13, Dec 1998.

[3] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications ACM*, 17(12):685–690, Dec. 1974.

[4] S. Adyanthaya, M. Geilen, T. Basten, R. Schiffelers, B. Theelen, and J. Voeten. Fast multiprocessor scheduling with fixed task binding of large scale industrial cyber physical systems. In *Euromicro Conference on Digital System Design (DSD)*, pages 979–988. IEEE, Sept 2013.

[5] S. Adyanthaya, M. Geilen, T. Basten, J. Voeten, and R. Schiffelers. Iterative robust multiprocessor scheduling. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS'15, pages 23–32, New York, NY, USA, 2015. ACM.

[6] S. Adyanthaya, M. Geilen, T. Basten, J. Voeten, and R. Schiffelers. Communication aware multiprocessor binding for shared memory systems. In *Proceedings of the 11th International Symposium on Industrial Embedded Systems*, SIES. IEEE, *Article in press*, 2016.

[7] S. Adyanthaya, Z. Zhang, M. Geilen, J. Voeten, T. Basten, and R. Schiffelers. Robustness analysis of multiprocessor schedules. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 9–17. IEEE, July 2014.

[8] A. Agarwal, D. Blaauw, and V. Zolotov. Statistical timing analysis for intra-die process variations with spatial correlations. In *Computer Aided Design, ICCAD-2003. International Conference on*, pages 900–907, 2003.

[9] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[10] S. Ali, A. Maciejewski, H. Siegel, and J.-K. Kim. Measuring the robustness of a resource allocation. *Parallel and Distributed Systems, IEEE Transactions on*, 15(7):630–641, July 2004.

[11] E. Artin. *The gamma function.* Holt, Rinehart and Winston, New York, 1964.

[12] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, Oct. 2010.

[13] S. Baruah and B. Brandenburg. Multiprocessor feasibility analysis of recurrent task systems with specified processor affinities. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 160–169, Dec 2013.

[14] S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese. A generalized parallel task model for recurrent real-time processes. In *RTSS*, 2012.

[15] M. Berkelaar. Statistical delay calculation, a linear time method. In *TAU (ACM/IEEE workshop on timing issues in the specification and synthesis of digital systems)*, December 1997.

[16] S. Bhardwaj, S. Vrudhula, and D. Blaauw. τau: Timing analysis under uncertainty. In *Computer Aided Design, ICCAD-2003. International Conference on*, pages 615–620, 2003.

[17] L. Bianchi, M. Dorigo, L. M. Gambardella, and W. J. Gutjahr. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, 8(2):239–287, 2009.

[18] L. Boloni and D. C. Marinescu. Robust scheduling of metaprograms. *Journal of Scheduling*, 5(5):395–412, 2002.

[19] B. B. Brandenburg, J. M. Calandrino, and J. H. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proceedings of the 2008 Real-Time Systems Symposium*, RTSS '08.

[20] H. Butler. Position control in lithographic equipment [applications of control]. *Control Systems, IEEE*, 31(5):28–47, Oct 2011.

[21] A. C. Cameron and F. A. Windmeijer. An r-squared measure of goodness of fit for some common nonlinear regression models. *Journal of Econometrics*, 77(2):329 – 342, 1997.

[22] L.-C. Canon and E. Jeannot. Evaluation and optimization of the robustness of dag schedules in heterogeneous environments. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):532–546, 2010.

[23] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176, 1989.

[24] B. Cirou and E. Jeannot. Triplet: A clustering scheduling algorithm for heterogeneous systems. In *International Conference on Parallel Processing Workshops*, pages 231–236, 2001.

[25] C. E. Clark. The greatest of a finite set of random variables. *Operations Research*, 9(2):145–162, 1961.

[26] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[27] M. Cosnard, M. Marrakchi, Y. Robert, and D. Trystram. Parallel gaussian elimination on an mimd computer. *Parallel Computing*, 6(3):275 – 296, 1988.

[28] R. L. Daniels and J. E. Carrillo. β-robust scheduling for single-machine systems with uncertain processing times. *IIE Transactions*, 29(11):977–985, 1997.

[29] A. Davenport, C. Gefflot, and C. Beck. Slack-based techniques for robust schedules. In *Sixth European Conference on Planning*, 2014.

[30] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1–35:44, Oct. 2011.

[31] J. Engblom. Analysis of the execution time unpredictability caused by dynamic branch prediction. In *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*, pages 152–159, May 2003.

[32] D. England, J. B. Weissman, and J. Sadagopan. A new metric for robustness with application to job scheduling. In *HPDC*, pages 135–143. IEEE, 2005.

[33] F. Fauberteau et al. Robust partitioned scheduling for real-time multiprocessor systems. In *Distributed, Parallel and Biologically Inspired Systems*, pages 193–204. Springer, 2010.

[34] A. Forti. *DAG scheduling in grid computing systems*. PhD thesis, University of Udine, 2006.

[35] R. Frijns. *Platform-based Design for High-Performance Mechatronic Systems.* PhD thesis, Eindhoven University of Technology, 2015.

[36] R. M. W. Frijns, S. Adyanthaya, S. Stuijk, J. P. M. Voeten, M. C. W. Geilen, R. R. H. Schiffelers, and H. Corporaal. Timing analysis of first-come first-served scheduled interval-timed directed acyclic graphs. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, pages 288:1–288:6. IEEE, March 2014.

[37] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[38] F. Glover. Tabu search – part i. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[39] P. E. Greenwood and M. S. Nikulin. *A guide to chi-squared testing.* Wiley-Interscience, 1996.

[40] D. Hardy and I. Puaut. Static probabilistic worst case execution time estimation for architectures with faulty instruction caches. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, pages 35–44, New York, NY, USA, 2013. ACM.

[41] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[42] C. J. Hughes, P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan. Variability in the execution of multimedia applications and implications for architecture. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 254–265, New York, NY, USA, 2001. ACM.

[43] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley-IEEE Computer Society Pr, 2008.

[44] D. Khaldi, P. Jouvelot, and C. Ancourt. Parallelizing with BDSC, a resource-constrained scheduling algorithm for shared and distributed memory systems. *Parallel Computing*, 41:66 – 89, 2015.

[45] M. A. Khan. Scheduling for heterogeneous systems using constrained critical paths. *Parallel Computing*, 38(45):175–193, 2012.

[46] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors.* IEEE Computer Society, 1997.

[47] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[48] B. Kruatrachue and T. Lewis. Grain size determination for parallel processing. *IEEE Software*, 5(1):23–32, Jan 1988.

[49] Y. K. Kwok and I. Ahmad. Benchmarking the task graph scheduling algorithms. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998*, pages 531–537, Mar 1998.

[50] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, Dec. 1999.

[51] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 239–248, July 2009.

[52] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept 1987.

[53] K. Li et al. Scheduling precedence constrained stochastic tasks on heterogeneous cluster systems. *Computers, IEEE Transactions on*, 64(1):191–204, Jan 2015.

[54] C. Liu and J. Anderson. Supporting graph-based real-time applications in distributed systems. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2011.

[55] M. Lombardi et al. Robust scheduling of task graphs under execution time uncertainty. *Computers, IEEE Transactions on*, 62(1):98–111, 2013.

[56] L. Lukasiak and A. Jakubowski. History of semiconductors. *Journal of Telecommunications and Information Technology*, (1):3–9, 2010.

[57] G. Moore. Progress in digital integrated electronics. In *Electron Devices Meeting, 1975 International*, volume 21, pages 11–13, 1975.

[58] S. Nadarajah and S. Kotz. Exact distribution of the max/min of two gaussian random variables. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(2):210–212, Feb 2008.

[59] C. Papadimitriou and M. Yannakakis. Scheduling interval-ordered tasks. *SIAM Journal on Computing*, 8(3):405–409, 1979.

[60] J. Parkhurst, J. Darringer, and B. Grundmann. From single core to multi-core: Preparing for a new exponential. In *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, pages 67–72, Nov 2006.

[61] A. Renyi. On the central limit theorem for the sum of a random number of independent random variables. *Acta Mathematica Academiae Scientiarum Hungarica*, 11(1-2):97–102, 1963.

[62] R. Rubinstein and D. Kroese. *Simulation and the Monte Carlo Method*. Wiley Series in Probability and Statistics. Wiley, 2008.

[63] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer. Modeling performance variation due to cache sharing. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 155–166, Feb 2013.

[64] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989.

[65] R. R. H. Schiffelers, W. Alberts, and J. P. M. Voeten. Model-based specification, analysis and synthesis of servo controllers for lithoscanners. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, MPM '12, pages 55–60, New York, NY, USA, 2012. ACM.

[66] R.-H. M. Schmidt. Ultra-precision engineering in lithographic exposure equipment for the semiconductor industry. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 370(1973):3950–3972, 2012.

[67] V. Shestak, J. Smith, A. A. Maciejewski, and H. J. Siegel. Stochastic robustness metric and its use for static resource allocations. *J. Parallel Distrib. Comput.*, 68(8):1157–1173, Aug. 2008.

[68] Z. Shi, E. Jeannot, and J. Dongarra. Robust task scheduling in non-deterministic heterogeneous computing systems. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–10, 2006.

[69] G. Sih and E. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, Feb 1993.

[70] M. Solar and M. Inostroza. A scheduling algorithm to optimize real-world applications. In *Distributed Computing Systems Workshops. Proceedings. 24th International Conference on*, pages 858–862, March 2004.

[71] F. Stork. *Stochastic resource-constrained project scheduling*. PhD thesis, Technische Universität Berlin, 2001.

[72] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, 2006.

[73] B. Theelen. *Performance Modelling for System-Level Design.* PhD thesis, Eindhoven University of Technology, 2004.

[74] B. Theelen. A performance analysis tool for scenario-aware streaming applications. In *Quantitative Evaluation of Systems, 2007. QEST 2007. Fourth International Conference on the*, pages 269–270, Sept 2007.

[75] A. H. Timmer. *From Design Space Exploration to Code Generation.* PhD thesis, Eindhoven University of Technology, 1996.

[76] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[77] M. A. van den Brink, H. Jasper, S. D. Slonaker, P. Wijnhoven, and F. Klaassen. Step-and-scan and step-and-repeat: a technology comparison. In *Proceedings of SPIE*, volume 2726, pages 734–753, 1996.

[78] M. Van Hauwermeiren and D. Vose. *A Compendium on Distributions.* [ebook]. Vose Software, Ghent, Belgium, 2009.

[79] J. Verriet. The complexity of scheduling typed task systems with and without communication delays. Technical report, Utrecht University, 1998.

[80] J. H. Verriet. Scheduling with communication for multiprocessor computation. Technical report, Utrecht University, 1998.

[81] J. P. M. Voeten, P. van der Putten, M. Geilen, and M. P. J. Stevens. Formal modelling of reactive hardware/software systems. In *J.P. Veen, Ed., Proceedings of ProRISC/IEEE 97, Utrecht : STW, Technology Foundation*, pages 663–670, 1997.

[82] D. Vose. *Risk analysis : a quantitative guide.* Wiley, Chichester, England, Hoboken, NJ, 2008.

[83] D. Wang, B. Gong, and G. Zhao. Estimating deadline-miss probabilities of tasks in large distributed systems. In R. Li, J. Cao, and J. Bourgeois, editors, *Advances in Grid and Pervasive Computing*, volume 7296 of *Lecture Notes in Computer Science*, pages 254–263. Springer Berlin Heidelberg, 2012.

[84] M.-Y. Wu and D. Gajski. Hypertool: a programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, July 1990.

[85] M.-Y. Wu, W. Shu, and J. Gu. Efficient local search for DAG scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 2001:617–627, 2001.

[86] T. Yang and A. Gerasoulis. PYRROS: Static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th International Conference on Supercomputing*, ICS '92, pages 428–437, 1992.

[87] T. Yang and A. Gerasoulis. DSC: scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, Sep 1994.

List of Abbreviations

BDSC Bounded Dominant Sequence Clustering
CALAP Classical As-Late-As-Possible
CCR Communication to Computation Ratio
CMA Clustering, Merging, Allocation
DAG Directed Acyclic Graph
DCS Divide & Conquer Search
DSC Dominant Sequence Clustering
DSL Domain Specific Language
ECF Earliest Classical As-Late-As-Possible First
EDDF Earliest Due-date First
EDF Earliest Deadline First
EUV Extreme Ultraviolet
FCFS First-Come-First-Served
FFT Fast Fourier Transforms
FJ Fork Join
GE Gaussian Elimination
IHRIF Iterative Highest Robustness Impact First
IO Input Output
MDC Molecular Dynamics Code
PERT Project Evaluation & Review Technique
POB Projection Optics Box
POOSL Parallel Object-Oriented Specification Language
SADF Scenario Aware Data Flow

List of Symbols

Notation	*Description*
a	Task
(a, b)	A dependency from task a to b not considering communication cost
$(a, b, \hat{c})$	A dependency from task a to b with a communication cost of $\hat{c}$
$\|f\|$	L^2 norm of a function f
$< f, g >$	L^2 inner product of two functions f and g
$before(r, a)$	Last task scheduled before task a on resource r
$blevel_a$	Bottom level of a task a
c_a	Completion time of a task a
c_g	Completion time of a gap g
CA	Cluster Allocation
CL	Set of clusters
cl	Cluster
CL	Set of clusters
CLD	Set of cluster dependencies
CT	Set of clustered Tasks
d_a	Deadline of a task a
D	Set of dependencies
$\mathbb{D}$	Probability density functions
dd_a	Due-date of a task a
dd_A	Due-date of a set A of tasks
DS_a	Dependent set of a task a
e_a	Execution time of a task a
ET	Set of enabled tasks
G	DAG
g	Gap
HCC	Highly Connected Cluster Function

Notation	*Description*
I_a^S	Impact of a task a in a schedule S
IE_a	Impact estimate of a task a
$IH_a(i)$	Impact history of a task a in the ith iteration of the iterative robust scheduler
$last_r$	Last task scheduled of resource r
$lastPred_a$	Last completion predecessor of a task a
$M_{<p,h>}$	Curve fitting metric of a pert distribution p on a histogram h
m	Makespan
max	Parameter 'maximum' of a PERT distribution
min	Parameter 'minimum' of a PERT distribution
$mode$	Parameter 'mode' or peak of a PERT distribution
$\mathbb{P}[x < y]$	Probability of x being less than y
P_{c_a}	Probability density function of the completion time of a task a
P_{e_a}	Probability density function of the execution time of a task a
P_{s_a}	Probability density function of the start time of a task a
$prec_\gamma$	Precision parameter for divide and conquer search on γ
$prec_m$	Precision parameter for divide and conquer search on $mode$
$pred(a)$	Set of predecessors of a task a
$\mathbb{R}^{\geq 0}$	Non-negative real numbers
$\mathbb{R}$	Real numbers
r_a	Resource of a task a
R	Set of resources
R	Read operation
S^{prt}	Partial schedule
s_a	Start time of a task a
s_g	Start time of gap g
S	Static-order schedule
SD	Scheduling decisions of a scheduler
sl_a	Slack of a task a
${sl_{ab}}^S$	Slack between two tasks a and b in a schedule S
SP_{FM}	Feasible multiprocessor scheduling problem
ST	Set of scheduled tasks
$succ(a)$	Set of successors of a task a
T	Set of tasks
t	Time
$tlevel_a$	Top level of a task a
U	Update operation
var	Variance of a set of simulated samples
X	Random variable for the number of deadline misses in a schedule
$\mathbb{Z}$	Integers
$\#$	Number of
$\perp$	No resource assigned to task
$\delta(t)$	Dirac δ-function
Δ	Bin-width of a histogram
γ	Parameter 'gamma' or intensity parameter of a PERT distribution
μ	Mean of a set of simulated samples
τ	Delay

Curriculum Vitae

Shreya Adyanthaya was born on 21 December 1987 in Mangalore, India. She received her B.E. in Computer Science and Engineering from Nitte Mahalinga Adyanthaya Memorial Institute of Technology under the Visvesvaraya Technological University, India, in 2009. She obtained her MTech in Software Engineering from the Manipal University, India, and her MSc in Computer Science and Engineering from the Eindhoven University of technology in 2011, as a part of a dual degree exchange program. The focus of her MSc degree was on Formal Methods and during her master thesis project she performed the formal modelling and verification of the ASML wafer scanner systems in UPPAAL. She started her PhD in November 2011 in the Electronic Systems group at the Department of Electrical Engineering of Eindhoven University of Technology. This PhD project is a joint collaboration of TU/e with ASML and TNO-ESI. During the course of her PhD work, she developed several scheduling algorithms for ASML that are currently integrated into their wafer scanner systems. These contributions also led to several publications and the contents of this thesis. She currently works as a design engineer at ASML.

List of Publications

First author

Conference papers

[1] S. Adyanthaya, M. Geilen, T. Basten, J. Voeten, and R. Schiffelers. Communication Aware Multiprocessor Binding for Shared Memory Systems. In Proceedings of the *11th IEEE International Symposium on Industrial Embedded Systems, SIES.* IEEE, ***Article in press***, 2016.

[2] S. Adyanthaya, M. Geilen, T. Basten, J. Voeten, and R. Schiffelers. Iterative robust multiprocessor scheduling. In Proceedings of the *23rd International Conference on Real Time and Networks Systems, RTNS'15*, pages 23-32, New York, NY, USA, 2015. ACM.

[3] S. Adyanthaya, Z. Zhang, M. Geilen, J. Voeten, T. Basten, and R. Schiffelers. Robustness analysis of multiprocessor schedules. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV) 2014*, pages 9-17. IEEE, July 2014.

[4] S. Adyanthaya, M. Geilen, T. Basten, R. Schiffelers, B. Theelen, and J. Voeten. Fast multiprocessor scheduling with fixed task binding of large scale industrial cyber physical systems. In *Euromicro Conference on Digital System Design (DSD)*, pages 979-988. IEEE, Sept 2013.

Posters

[5] S. Adyanthaya, R.R.H. Schiffelers, R. Theunissen, C. van Huët, M.C.W. Geilen, R.M.W. Frijns, and J.P.M. Voeten, Multi-core communication-aware scheduler for CARM 2G, In *14th ASML Technology Conference*, poster, 2013.

Co-author

Journal papers

[6] S. Adyanthaya, H. Alizadeh Ara, J. Bastos, A. Behrouzian, R. Medina Sánchez, J. van Pinxten, B. van der Sanden, U. Waqas, T. Basten, H. Corporaal, R. Frijns, M. Geilen, M. Hendriks, D. Goswami, S. Stuijk, M. Reniers, and J. Voeten. xCPS: A tool to eXplore Cyber Physical Systems. *SIGBED Review.* ACM, ***Accepted for publication***, 2016.

Conference papers

[7] R.M.W. Frijns, S. Adyanthaya, S. Stuijk, J.P.M. Voeten, M.C.W. Geilen, R.R.H. Schiffelers, and H. Corporaal. Timing analysis of first-come first-served scheduled interval-timed directed acyclic graphs. In Proceedings of the *Conference on Design, Automation & Test in Europe (DATE)*, pages 288:1-288:6, IEEE, March 2014.

[8] S. Adyanthaya, H. Alizadeh Ara, J. Bastos, A. Behrouzian, R. Medina Sánchez, J. van Pinxten, B. van der Sanden, U. Waqas, T. Basten, H. Corporaal, R. Frijns, M. Geilen, D. Goswami, S. Stuijk, M. Reniers, and J. Voeten. xCPS: A tool to eXplore Cyber Physical Systems. In Proceedings of the *Workshop on Embedded and Cyber-Physical Systems Education, WESE'15*, pages 3:1-3:8, New York, NY, USA, 2015. ACM.