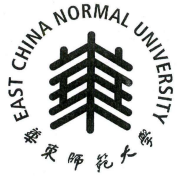
Fault-tolerant and energy-aware algorithms for workflows and real-time systems

Li Han

Numéro National de Thèse : 2020LYSEN013

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON

opérée par

l'École Normale Supérieure de Lyon

en cotutelle avec

East China Normal University

*École Doctorale N°*512

École Doctorale en Informatique et Mathématiques de Lyon

Spécialité : Informatique

présentée et soutenue publiquement le 06/05/2020, par :

Li HAN

Fault-tolerant and energy-aware algorithms for workflows and real-time systems

Algorithmes tolérants aux pannes et minimisant l'énergie pour les systèmes de tâches et les systèmes temps-réel

Devant le jury composé de :

Alix	MUNIER	Professeur, Univ. Paris 6	*Rapporteur*
Denis	TRYSTRAM	Professeur, INP Grenoble	*Rapporteur*
Brice	GOGLIN	Directeur de recherche, Inria Bordeaux	*Examinateur*
Veronika	SONIGO	Maître de Conférences, Univ. Franche Comté	*Examinatrice*
Robert	SPECK	Researcher, Jülich Supercomputing Center (Allemagne)	*Examinateur*
Jing	LIU	Professeur, ECNU (Chine)	*Co-tutrice de thèse*
Yves	ROBERT	Professeur, ENS de Lyon	*Directeur de thèse*
Frédéric	VIVIEN	Directeur de recherche, Inria Lyon	*Co-encadrant de thèse*

Contents

Introduction

In modern scientific research, the difficulty of solving problems by theory and experiment alone has gradually increased. Numerical methods have been used to simulate the physical world to solve complex problems. Computational science has become an indispensable tool for natural science research. In Oct. 2013, the Nobel Prize in Chemistry was awarded for "development of multiscale models for complex chemical systems". The selection committee pointed out that for today's chemists, computers have become an equally important tool as test tubes, and the simulation of life phenomena by computers has contributed to most of the research results of chemistry. This will undoubtedly further deepen the understanding and application of computational models and computational sciences in various disciplines, and will lead the development of these disciplines into the next wave.

The massive calculation requirements from scientific simulations have become the most direct driving force for the development of high-performance computers (HPC)[1], also known as supercomputers. The term refers to aggregating computing components in a way that produces much higher performance than those units working separately in solving large problems in science and engineering. It is widely used in aerospace aircraft design, energy exploration, long-term weather forecast, image processing, intelligence analysis, and other fields. All these areas are of great value to the development of national economy and national defense. Although the improvement of supercomputers drives new breakthroughs in numerous areas, it also brings new challenges. In February 2014, the ASCAC (Advanced Scientific Computing Advisory Committee) Subcommittee identified the top ten challenges [72] to achieve the development of an Exascale system (10^{18} floating point operation per second). This thesis addresses two prominent concerns in this list, namely, *resilience* and *energy efficiency*.

Resilience, or fault-tolerance, is stated as [72]: ensuring correct scientific computation in face of faults, reproducibility, and algorithm verification challenges. To fully develop the ability of a supercomputer (hundreds of thousands of cores), we have to divide a large problem into small ones, and compute each of them in parallel by different processing units (cores). As a result, the correctness of the final result is highly related to each single component. Indeed, with the advance of manufacturing, each individual resource becomes more reliable, but aggregating too many of them will result in frequent failures globally. Platform sizes have become so large that errors and failures are likely to strike at a high rate during application execution [24] which is for sure a barrier on the road to exascale. More precisely, the MTBF (Mean Time Between Failures) μ_P of the platform decreases linearly with the number of processors P, since

$$\mu_P = \frac{\mu_{\text{ind}}}{P}$$

where μ_{ind} is the MTBF of each individual component (see Proposition 1.2 in [56]). Take $\mu_{\text{ind}} = 10$ years as an example. If $P = 10^5$ then $\mu_P \approx 50$ minutes and if $P = 10^6$ then $\mu_P \approx 5$ minutes: from the point of view of fault-tolerance, scale is the enemy. Obviously, failure handling is critical for

[1] HPC could also refer to high performance computing (parallel computing), which could be clearly identified through different contexts.

highly parallel applications that use a large number of components for a significant amount of time, because such applications are likely to experience at least one failure during execution. If there is no fault-tolerance mechanism, once an error occurs, one needs to spend lots of time to re-execute, then the execution may be infinite.

In the literature, there are three terms frequently used when studying resilience: *fault*, *error* and *failure*. To clarify the differences between the definition of those terms [68]: a failure means that the external behavior is incorrect, where a system or component does not perform required function according to its specification. The failure occurred because the system is erroneous, while an error is the manifestation of a fault. In other words, a fault creates a latency error, which becomes effective when it is activate; when the error affects delivered services, a failure occurs.[2] Faults are mainly classified into two categories: permanent faults and transient faults.

Permanent faults are usually induced by hardware problems, e.g., oxide wearout, power loss, etc., which lead to fail-stop errors (hard errors) [77]. It remains for indefinite periods till corrective action, e.g., a part replacement, is taken. Once a fail-stop error occurs, the execution of the processor stops immediately, all contents in its memory is lost, and the computations have to be restarted from scratch, either on the same processor once it reboots or on a spare. The de-facto approach to handle such failures is Checkpoint/Restart (C/R), by which application state is saved to stable storage, such as a shared file system, throughout execution.

Transient faults are usually manifested as bit-flips and due to any of the following causes: radiation, minimum voltage, or thermal cycling, etc., which lead to soft errors [108]. Soft errors cause a task execution to fail but without completely losing the data present in the processor memory. The most insidious form of soft error is silent data corruption (SDC), also known as silent errors, where the fault induces erroneous output without interrupting the execution. To avoid silent errors, designers need to deploy error detection mechanism. By applying the silent error detector, the execution is interrupted once an error is detected, thereby we avoid generating incorrect outputs. But this does not reduce the error rate nor correct the error. We call errors in this category detected unrecoverable errors (DUE). Local checkpointing (or more precisely making a copy of all task input/output data), and/or task replication, are the most widely used technique to recover from soft errors.

We can see that fault-tolerant strategies typically require redundancy of system resources, either continuously (*replication*) or at periodic intervals (migration from faulty node to spare node, rollback and recovery). In the latter case, the state of an application must be preserved (*checkpointing*), and the system must roll back to the last saved checkpoint. However the amount of replication and/or the frequency of checkpointing must be optimized carefully. For example, checkpointing frequently degrades performance, cause the system takes time to save files instead of doing computation. On the other hand, the application is at risk between two checkpoints, and the longer the checkpoint interval, the larger the penalty paid for re-executing work after a failure. We will deal with the trade-off between performance (in terms of the total execution time) and the resilience of fail-stop errors in Part I.

Redundant computations and/or communications called by the fault-tolerant methods, not only degrade the system performance, but also conflict with the energy minimization objective. Power management is necessary due to both monetary and environmental constraints. Presently, large computing centers are among the largest consumers of energy. Energy is needed to provide power to the individual cores and also to provide cooling for the system. On the 15th November 2007, the list of Green500

[2] Although there are slight differences in definition of those three terms, we only focus on their consequences or effects on the execution and systems. So in the rest of this thesis, we use fault, error and failure indifferently.

was announced at SC07 [70]. As a complement to the rank TOP500, the Green500 list ranks the supercomputers in the world by energy efficiency (FLOPS-per-Watt), which shows the trend that we do not only focus on performance-at-any-cost supercomputers. Sustainable supercomputing is the new era. Dynamic voltage and frequency scaling (DVFS) is a widely used technique to manage energy consumption, but it can increase execution time and severely degrade performance. DVFS is a mechanism that allows to adjust the CPU voltage and frequency; this is a well-known technique that trades off the processing speed with energy savings. In the context of reliability, DVFS has a negative impact on the transient fault rate [33]: as we decrease the supply voltage and frequency to save power, the transient fault rate significantly increases. As a consequence, when we scale down the frequency level, we must take into account the reliability and performance degradation. We will dig into the interplay between energy, reliability and performance in the context of transient errors in Part II.

The main goal of this thesis is to re-design scheduling algorithms and to investigate trade-offs between multiple criteria (i.e., system performance, resilience and energy consumption). *Scheduling* problems have been well studied: given a set of tasks and resources, how do we allocate them and in which order should tasks assigned on the same core to be executed? With different objectives into play, the problem becomes even more difficult. This is a challenging but unavoidable multi-criteria optimization problem, whose solution is critical for many applications and large-scale systems.

In Part I, we focus on task graphs (workflows) scheduling and checkpointing strategies for fail-stop errors, starting with an application task graph deployed on a large-scale platform. With resilience into play, which tasks should we checkpoint in order to minimize the total execution time? The objective is to design optimal solutions for special classes of task graphs, namely M-SPG in Chapter 2, and to provide general-purpose heuristics for arbitrary ones in Chapter 3. Then in Part II, we will consider several application instances running on the platform, which means that tasks are periodically input to the platform. This is in the context of real-time scheduling. Traditionally, replication is used to mitigate the impact of transient errors. We investigate the number of replicas that are needed to achieve the reliability goal for each task, with interplay between time constrains and energy minimization, to which processors should these replicas be allocated and in which order should they be scheduled? This study is done first for homogeneous systems in Chapter 5 and then for heterogeneous systems in Chapter 6. The main contributions of each chapter are summarized as below.

Part I

This part starts with a short introduction and related work in the context of scheduling and checkpointing workflows in Chapter 1.

Chapter 2: Optimal solutions for special classes of task graphs [C2, J1]

In this chapter, we consider the problem of orchestrating the execution of workflow applications structured as Directed Acyclic Graphs (DAGs) on parallel computing platforms that are subject to fail-stop failures. The objective is to minimize expected overall execution time, or makespan. A solution to this problem consists of a schedule of the workflow tasks on the available processors and of a decision of which application data to checkpoint to stable storage, so as to mitigate the impact of processor failures. To address this challenge, we first consider in this chapter, a restricted class of graphs, Minimal Series-Parallel Graphs (M-SPGs), which is relevant to many real-world workflow applications. For this class of graphs, we propose a recursive list-scheduling algorithm that exploits the M-SPG structure to assign sub-graphs to individual processors, and uses dynamic programming to decide optimally how to

checkpoint these sub-graphs. We assess the performance of our algorithm for production workflow configurations, comparing it to (i) an approach in which all application data is checkpointed (CKPTALL), which corresponds to the standard way in which most production workflows are executed today; and (ii) an approach in which no application data is checkpointed (CKPTNONE). Results demonstrate that our algorithm outperforms both the former approach, because of lower checkpointing overhead, and the latter approach, because of better resilience to failures.

Chapter 3: Generic approaches for arbitrary task graphs [C3, J2]

After studying the optimal solutions for the specific type of graphs, namely M-SPGS (Minimal Series-Parallel Graphs), in Chapter 2, in this following chapter, we deal with scheduling and checkpointing strategies to execute general scientific workflows (task graphs) on failure-prone large-scale platforms. To the best of our knowledge, this work is the first to target fail-stop errors for arbitrary workflows. Most previous work addresses soft errors, which corrupt the task being executed by a processor but do not cause the entire memory of that processor to be lost (contrarily to fail-stop errors), or only consider a restricted type of graphs (e.g., a chain of tasks). We revisit classical mapping heuristics such as HEFT and MINMIN and complement them with several checkpointing strategies. The objective is still to derive an efficient trade-off between checkpointing every task (CKPTALL), which is an overkill when failures are rare events, and checkpointing no task (CKPTNONE), which induces dramatic re-execution overhead even when only a few failures strike during execution. Contrarily to previous work, the new approach applies to arbitrary workflows, not just special classes of dependence graphs such as a chain of tasks or M-SPGS. Extensive experiments report significant gain over both CKPTALL and CKPTNONE, for a wide variety of workflows.

Part II

This part starts with a short introduction and related work in the context of scheduling real-time applications on both homogeneous and heterogeneous systems in Chapter 4.

Chapter 5: Homogeneous platforms [C4]

This chapter revisits the real-time scheduling problem on homogeneous platforms recently introduced by Haque, Aydin and Zhu (2017). In this challenging problem, task redundancy ensures a given level of reliability while incurring a significant energy cost. The inputs to the problem are: a set of real-time tasks, a set of identical processors and a reliability target. Our goal is to find the replica settings that guarantee the target reliability threshold and meet all deadlines, while minimizing energy consumption. By carefully setting processing frequencies, allocating tasks to processors and ordering task executions, we improve on the previous state-of-the-art approach with an average gain in energy of 20%. Furthermore, we establish the first complexity results for specific instances of the problem.

Chapter 6: Heterogeneous platforms

Following the study in Chapter 5, in this chapter we focus on heterogeneous platforms, which are composed of processors with different (and possibly unrelated) characteristics, including speed profile, energy cost and failure rate. We provide several mapping and scheduling heuristics towards the challenging multi-criteria optimization problem: minimize the expected energy consumption while enforcing the reliability threshold and meeting all task deadlines. Compared to homogeneous platforms, heterogeneity

complicates the problem as we could not know the number of replicas needed for each task to meet its reliability threshold before deciding its processor. For each task, we have to decide: (i) how many replicas to use; (ii) on which processor to map each replica; and (iii) when to schedule each replica on its assigned processor. Different mappings achieve different levels of reliability and consume different amounts of energy. Scheduling matters because once a task replica is successful, the other replicas of that task are cancelled, which calls for minimizing the amount of temporal overlap between any replica pair. The experiments are conducted for a comprehensive set of execution scenarios, with a wide range of processor speed profiles and failure rates. The comparison results reveal that our strategies perform better than the random baseline, with a gain of 40% in energy consumption, for nearly all cases. The absolute performance of the heuristics is assessed by a comparison with a lower bound; the best heuristics achieve an excellent performance, with an average value only 4% higher than the lower bound.

Résumé français

En février 2014, l'ASCAC (Advanced Scientific Computing Advisory Committee) sous-comite a identifie les dix principaux défis pour parvenir au développement d'un système Exascale (plates-formes pouvant effectuer 10^{18} operations par secondes). Cette thèse se concentre sur deux problèmes majeurs de cette liste, dans le contexte du calcul haute performance (HPC): la résilience et la consommation d'énergie.

Pour fournir la puissance de calcul requise par la recherche scientifique moderne, le nombre d'unités de calcul dans les super-ordinateurs a considérablement augmenté ces dernières années, ce qui entraîne une augmentation de la fréquence des erreurs. Plus précisément, le temps moyen entre deux fautes (MTBF) est proportionnel au MTBF de ses composants, mais aussi a l'inverse du nombre de processeurs sur ces machines. Alors que la fiabilite des composants pris de maniere independante augmente, leur nombre augmente également, et de maniere exponentielle. Par consequent, ce MTBF decroit rapidement. De toute évidence, la mise en œuvre de mécanismes de tolérance aux pannes, afin d'assurer la résilience des applications, devient critique pour les applications hautement parallèles qui utilisent un grand nombre de composants pendant une période de temps significative. En effet, en l'absence de tout mécanisme de tolérance aux pannes, une application pourrait être ne jamais terminer.

D'un autre côté, il est nécessaire de minimiser la consommation énergétique à cause des contraintes budgétaires et environnementales. Actuellement, les grands centres de calcul comptent parmi les plus gros consommateurs d'énergie. L'énergie est nécessaire pour alimenter les cœurs individuels et également pour refroidir le système. Ceci est d'autant plus important que la résilience nécessite souvent une redondance dans le temps (points de sauvegardes) et/ou dans l'espace (réplication), qui induit une consommation énergétique supplémentaire. Par ailleurs, certaines des technologies qui réduisent la consommation d'énergie (e.g., DVFS) ont des effets négatifs sur les performances et la résilience. Par conséquent, lorsque nous gérons l'énergie consommée, nous devons prendre en compte la fiabilité et la dégradation des performances.

Dans ce contexte, nous concevons des algorithmes d'ordonnancement pour étudier les compromis entre performance, résilience et consommation d'énergie. Les problèmes d'ordonnancement ont été bien étudiés: étant donné un ensemble de tâches et de ressources, comment pouvons-nous les allouer et dans quel ordre les tâches assignées sur le même processeur doivent-elles être exécutées? Avec différents objectifs en jeu, le problème devient encore plus difficile. Il s'agit d'un problème d'optimisation multi-critère difficile mais inévitable, dont la solution est critique pour de nombreuses applications et systèmes à grande échelle.

La Partie I de cette thèse se concentre sur l'ordonnancement des graphes de tâches sujets à des pannes. La question est alors de décider quelle tâche doit être sauvegardée (redondance dans le temps) afin de minimiser le temps d'exécution total. L'objectif est alors de concevoir des solutions optimales pour des classes spéciales de graphes de tâches (Chapitre 2) et de fournir des heuristiques pour le cas général (Chapitre 3). Nous considérons dans la Partie II l'ordonnancement d'ensembles de tâches périodiques indépendantes dans un contexte temps-réel, sujet à des erreurs silencieuses. Nous étudions

combien de répliques sont nécessaires (redondance en espace) et analysons l'interaction entre dates butoir, fiabilité, et minimisation d'énergie. Cette étude est effectuée d'abord pour les systèmes homogènes au Chapitre 5, puis pour les systèmes hétérogènes au Chapitre 6. Les contributions principales de chaque chapitre sont resumees ci-dessous.

Partie I:

Cette partie commence par une brève introduction et des travaux connexes dans le contexte de l'ordonnancement et le checkpoint de graphes de tâches dans le Chapitre 1.

Chapitre 2: Solutions optimales pour des classes spéciales de graphes de tâches [C2, J1]

Ce chapitre considère l'ordonnancement de workflows (applications structurées en forme de graphes de tâches acycliques, ou DAGs) sur des plates-formes parallèles à grande échelle, soumises à des erreurs fatales. L'objectif est de minimiser l'espérance du temps total d'exécution, ou makespan. Une solution à ce problème comprend l'allocation ordonnée des tâches aux processeurs, et les décisions de checkpoint: quelles tâches sont suivies d'un checkpoint? Même pour une solution donnée, le calcul du makespan reste difficile. Nous nous restreignons d'abord à une classe de DAGs particuliers, les graphes séries-parallèles minimaux, ou M-SPGs. De nombreux workflows issus des applications ont pour graphe un M-SPG. Pour de tels graphes, nous proposons un algorithme qui utilise la structure récursive du M-SPG pour allouer des sous-graphes à chaque processeur, et utilise la programmation dynamique pour décider quelles tâches checkpointer. Il est alors possible de calculer efficacement le makespan via des algorithmes d'évaluation de DAGs probabilistes à deux états. Nous établissons expérimentalement la performance de notre approche en la comparant, sur des workflows applicatifs bien connus, avec l'approche qui checkpointe toutes les tâches et celle qui n'en checkpointe aucune. Les résultats montrent que notre approche réalise un bon compromis entre les deux approches extrêmes, avec moins de surcoût de checkpoint que la stratégie qui checkpointe tout le temps, et une meilleure résilience que celle qui ne checkpointe jamais. A notre connaissance, notre approche est la première à considérer des DAGs plus généraux que des chaînes pour les erreurs fatales.

Chapitre 3: Une approche générique pour les graphes de tâches arbitraires [C3, J2]

A partir des resultats obtenus dans le Chapitre 2, qui se concentre sur des M-SPGs, ce chapitre etends et generalise les stratégies pour le cas général: l'ordonnancement et les stratégies de checkpoint utiles à l'exécution d'applications scientifiques structurées en forme DAGs généraux, sur des plateformes à grande échelle, sujettes aux fautes. A notre connaissance, ce travail est le premier à traiter des erreurs fatales pour des graphes de tâches arbitraires. La plupart des travaux existants traitent des erreurs silencieuses, qui corrompent la tâche en train d'être exécutée sur un processeur mais ne provoquent pas la disparition totale de la mémoire de ce processeur, contrairement aux erreurs fatales. Nous revisitons les heuristiques d'allocation classiques telles que HEFT et MINMIN, auxquelles nous rajoutons plusieurs stratégies de checkpoint. L'objectif est comme au chapitre precedent de trouver un juste milieu efficace entre checkpointer toutes les tâches (CKPTALL), ce qui est trop lourd quand les erreurs surviennent rarement, et n'en checkpointer aucune (CKPTNONE), ce qui induit des temps de ré-exécution élevés, même quand seulement quelques fautes surgissent durant l'exécution. Contrairement à ce qui a été fait précédemment, notre approche s'applique à des graphes de tâches quelconques, pas seulement à certaines classes spéciales de graphes de tâches comme les M-SPGs (Graphe Série-Parallèle Minimal) ou

les chaînes. Plusieurs expériences montrent un gain significatif par rapport à CKPTALL et CKPTNONE, pour une large variété de graphes de tâches.

Partie II:

Cette partie commence par une brève introduction et des travaux connexes dans le contexte de l'ordonnancement d'applications temps-réel sur les deux systèmes homogènes et hétérogènes dans le Chapitre 4.

Chapitre 5: Les plates-formes homogènes [C4]

Ce chapitre s'intéresse à un problème d'ordonnancement en temps-réel sur des plates-formes homogènes étudié par Haque, Aydin and Zhu et récemment paru dans IEEE TPDS [54]. Dans ce problème difficile, la redondance des tâches garantit un niveau de fiabilité donné tout en entraînant un consommation énergétique considérable. Les entrées du problème sont: un ensemble de tâches temps-réel, un ensemble de processeurs identiques et un objectif de fiabilité. L'objectif est de concevoir des nouvelles stratégies qui garantissant la fiabilité cible seuil et respecter tous les dates butoir, tout en minimisant la consommation d'énergie. En définissant soigneusement les fréquences de traitement, en allouant des tâches aux processeurs et en commandant la tâche exécutions, nous améliorons significativement les résultats de la précédente approche avec un gain d'énergie moyen de 20%. De plus, nous établissons les premiers résultats de complexité pour des cas particuliers du problème.

Chapitre 6: Les plates-formes heterogenes

Contrairement au chapitre précédent qui etudie des plates-formes composees de processeurs identiques, ce travail se concentre sur les plates-formes heterogenes. Comme motive plus haut, les plates-formes modernes comportent de plus en plus d'unites de calcul dediees, qui sont composées de processeurs ayant des caractéristiques différentes (et éventuellement non liées), y compris profil de vitesse, coût énergétique et taux d'erreur. Nous fournissons plusieurs heuristiques de placement et d'ordonnancement vers le difficile problème d'optimisation multi-critère: minimiser la consommation d'énergie attendue tout en respectant le seuil de fiabilité et en respectant toutes les échéances des tâches. Par rapport aux plates-formes homogènes, l'hétérogénéité complique le problème car on ne connait pas le nombre de répliques nécessaires pour que chaque tâche atteigne son seuil de fiabilité avant de décider sur quels processeurs exécuter ces répliques. Pour chaque tâche, nous devons décider: (i) combien de répliques utiliser; (ii) sur quel processeur allouer chaque réplique; et (iii) quand planifier chaque réplique sur son processeur attribué. Différents placement atteignent différents niveaux de fiabilité et consomment différentes quantités d'énergie. L'ordonnancement est important car une fois qu'une réplique de tâche réussit, les autres répliques de cette les tâches sont annulées, ce qui nécessite de minimiser la quantité de chevauchement temporel entre n'importe quelle paire de répliques. Les expériences sont menées pour un ensemble complet de scénarios d'exécution, avec une large gamme de profils de vitesse de processeur et de taux d'erreur. Les résultats de la comparaison révèlent que nos stratégies fonctionnent mieux que la ligne de base aléatoire, avec un gain de 40% de consommation d'énergie, dans presque tous les cas. La performance absolue de l'heuristique est évaluée par une comparaison avec une borne inférieure; la meilleure heuristique atteint une excellente performance, avec une valeur moyenne de seulement 4% supérieure à la borne inférieure.

Part I

Scheduling and checkpointing workflows for fail-stop errors

Chapter 1

Framework

1.1 Introduction

In the first part of this thesis, we deal with scheduling techniques to deploy scientific workflows on large parallel or distributed platforms subject to fail-stop processor failures, e.g., a large-scale cluster. Scientific workflows, also named *task graphs*, are the archetype of HPC (High Performance Computing) applications, which are naturally partitioned into tasks that represent computational kernels. The tasks are partially ordered because the output of some tasks may be needed as input to some other tasks. Altogether, the application is structured as a DAG (Directed Acyclic Graph) whose nodes are the tasks and whose edges enforce the dependencies. Nodes are weighted by the computational requirements (in flops) while edges are weighted by the size of communicated data (in bytes). Given a workflow and a platform, the problem of mapping the tasks onto the processors and to schedule them so as to minimize the total execution time, or *makespan*, has received considerable attention in the past.

As stated in the Introduction section, considering the increasing number of failures, the classical mapping and scheduling problem has recently been revisited to account for the fact that errors can strike during execution. In the case of a fail-stop error (e.g., a crash due to a power loss or some other hardware problem), the execution of the processor stops, all the content of its memory is lost, and the computations have to be restarted from scratch, either on the same processor once it reboots or on a spare. The de-facto approach to handle such failures is Checkpoint/Restart (C/R), by which application state is saved to stable storage, such as a shared file system, throughout execution. Because workflows are structured as DAGs of tasks, they are good candidates for a C/R approach. First, tasks can be checkpointed individually and asynchronously. Second, rather than checkpointing the entire memory footprint of a task, it is only necessary to checkpoint its output data.

Checkpointing strategy adds redundancy in time, as intermediate results are stored in between task executions and computations can be re-executed later on. There is an obvious trade-off between the amount of time one is willing to spend on checkpoints, and the amount of time wasted in re-executions in case of errors. In this part, we consider the following problem: scheduling a workflow execution on failure-prone processors and deciding which task to checkpoint. The objective is to minimize the expectation of the execution time, or makespan.

The common strategy used in practice is *checkpoint everything*, or CKPTALL: all output data of each task is saved onto stable storage (in which case we say "the task is checkpointed"). For instance, in production Workflow Management Systems (WMSs) [1, 2, 32, 38, 109, 110], the default behavior is that all output data is saved to files and all input data is read from files, which is exactly the CKPTALL strategy. While this strategy leads to fast restarts in case of failures, its downside is that it maximizes checkpointing overhead. At the other end of the spectrum, it would be a *checkpoint nothing* strategy, or

CKPTNONE, by which all output data is kept in memory (up to memory capacity constraints) and no task is checkpointed. This corresponds to "in-situ" workflow executions, which has been proposed to reduce I/O overhead [119]. The downside is that, in case of a failure, a large number of tasks may have to be re-executed, leading to slow restarts. The objective of this part is to achieve a desirable trade-off between these two extremes.

In order to design algorithms for this optimization problem, we face two major difficulties stated as following:

- **Evaluation of expected makespan.** The makespan is a random variable because task execution times are probabilistic, due to failures causing task re-executions. The complexity of this problem is steep. In fact, the complexity of computing the expected makespan of a given solution is already difficult. A solution consists of an ordered list of tasks to execute for each processor; and for each task whether or not to save its output data to stable storage after its execution. In a failure-free execution, the makespan of a solution is simply the longest path in the DAG, accounting for serialized task executions at each processor. With failures, however, estimating the expected makespan of a solution is difficult. Consider the CKPTALL strategy and a solution in which each task is assigned to a different processor. Computing the expected makespan amounts to computing the expected longest path in the schedule. Unfortunately, computing the expected length of the longest path in a DAG with probabilistic task durations is a known difficult problem [46, 79]. Even in the simplified case when task durations are random variables that can take only two discrete values, the problem is #P-complete [46]. [1] As a result, several approximation methods have been developed to estimate the expected longest path of a DAG with probabilistic task durations, including Monte-Carlo simulations (see Section 2.2.5 for a detailed discussion). The complexity for computing the expected makespan for the CKPTNONE strategy is open, we proved its #P-completeness in Section 2.5.

- **Processor interference.** The aim of this part is to design a strategy that achieves better performance than CKPTALL and than CKPTNONE by checkpointing some, but not all, tasks. When all tasks are checkpointed, failures are contained since a task can just be restarted after a failure by reading input data that has been saved to stable storage by that task's checkpointed predecessors. This is no longer the case when some tasks are not checkpointed, which gives rise to new difficulties when trying to estimate the expected makespan (and thus minimize it). This is because a failure on a processor can lead to the re-execution of not only the failed task itself, but also uncheckpointed tasks on the same processor or on other processors. As a result, we can no longer approximate the makespan by Monte-Carlo simulations which sample independently the expected execution time of each task. In this case, a discrete-event simulator will be needed as we described in Section 3.3.2. Furthermore, identifying which tasks must be re-executed after a failure depends on how inter-processor communication is performed (e.g., at which time a task's output data is sent to another processor after that task completes, whether the sent data is kept in the memory of the sender and for how long). Sections 2.1 and 3.1 provide detailed examples that highlight these difficulties. For simplicity we refer to these difficulties as "*processor interference*" because a failure on a processor can cause task re-executions on other processors.

Our key observation is that a way of avoiding processor interference completely, and thus of designing of a strategy that checkpoints only some tasks, is to prohibit "*crossover dependencies*". We define a crossover dependency as a dependency between a task T and a direct successor T' that are scheduled on

[1] Recall that #P is the class of counting problems that correspond to NP decision problems [16, 81, 106], and that #P-complete problems are at least as hard as NP-complete problems.

different processors, where the output data of T is not checkpointed. Prohibiting crossover dependencies reduces the difficulty of our problem in four ways: 1) As discussed above and in Sections 2.1 and 3.1, with crossover dependencies, a few failures can lead to many task re-executions and data re-transfers, during which other failures can occur. Prohibiting crossover dependencies avoids such complex scenarios because failures are contained to a single processor, thus enabling simple task restarts; 2) Without crossover dependencies, there is no longer any need for inter-processor communications and thus for any assumptions regarding these communications. The local storage/memory at each processor is limited to storing data that is input to tasks that will execute on that same processor; 3) Without crossover dependencies, it is possible to determine the optimal set of tasks to checkpoint for groups of tasks assigned to a single processor (see the dynamic programming proposed in Section 2.4.2); 4) Without crossover dependencies, computing the expected makespan reduces, as for CKPTALL, to computing the longest path in a DAG with probabilistic task durations for M-SPGS.

To the best of our knowledge, no general solution is available for our problem. For specific applications that can be represented as a chain of n atomic tasks, Toueg and Babaoglu have proposed a polynomial dynamic programming algorithm whose complexity is $O(n^3)$ [102]. In the following of this part, we first present related work in Section 1.2, then we are going to explore more generic approaches, starting with designing optimal solutions for M-SPGS in Chapter 2, where we take advantages of the recursive structure of M-SPGS. Then in Chapter 3, we extend these results for arbitrary DAGs.

1.2 Related work

Checkpointing workflows has received considerable attention in the recent years, but no satisfactory solution has been proposed for fail-stop failures and general DAGs. For completeness we first review related work devoted to soft errors (Section 1.2.1). We then review work devoted, like this work, to fail-stop errors (Section 1.2.2). We conclude with some comments on branch-and-bound methods for scheduling, and their possible extension to include checkpoints (Section 1.2.3)

1.2.1 Soft and silent errors

Many authors have considered soft errors, by which a task execution fails but does not lead to completely losing the data present in the processor memory. Local checkpointing, or more precisely making a copy of all task input/output data, is the most widely used technique to address soft errors. If a soft error occurs during its execution, a task can then be re-executed. This solution can be too costly, and it is possible to save a copy of task input/output data only periodically, at the price of more re-execution when an error is detected. This is the trade-off analyzed by Cao et al. [21] for Cholesky factorization. Several authors have suggested techniques that identify tasks on the critical path, and then making scheduling decisions that attempt to ensure the timely execution of these tasks [63, 66]. Another widely used technique to cope with soft errors is task replication, the challenge being to avoid over-duplicating tasks so as to strike a good balance between fast failure-free executions and resilient executions [27]. Two representative practical frameworks are the NARBIT system [67], which recovers from soft errors via task replication and work stealing, and Nanos [75, 97], a runtime system that supports the OpenMP programming model.

Silent errors represent a different challenge as they do not interrupt the execution of the task but corrupt its output data. However, their net effect is the same as other soft errors, since a task must be re-executed whenever a silent error is detected (a silent error detector is applied at the end of a task's execution). Local checkpointing (making copies of input/output data) or replicating tasks and comparing

outputs, are the two common techniques to mitigate the impact of silent errors. With checkpointing, several application-specific detectors can be used to avoid replication and increase performance in failure-free executions. Two well-known examples are Algorithm-Based Fault Tolerance (ABFT) [17, 59, 90] and silent error detectors based on domain-specific data analytics [9, 10, 12]. As we only consider fail-stop errors in this part, we do not need to use fault detectors.

Several studies exist that attempt to provide resilience to arbitrary DAGs in the presence of silent errors. All of them are based on some task replication mechanism. Hashimoto et al. [55] propose two multiprocessor scheduling algorithms for arbitrary DAGs, but they can only work on systems victim of at most one single silent error. The other works we are aware of try to maximize reliability, that is, the probability that the application execution is not victim of a single failure. Girault and Kalla [42] propose an exponential-time algorithm for bi-criteria multiprocessor scheduling which returns a static schedule for the input DAG under upper bound constraints on the application execution time and on the *global system failure rate*. Subasi et al. [96] use partial replication to improve the reliability of an application in presence of silent and fail-stop errors. Works that optimize reliability do not guarantee that all executions will eventually succeed (because, for instance, not all failure patterns are covered by the chosen replication scheme). By contrast, works, like this one, that optimize the expectation of the makespan guarantee that all executions successfully complete (otherwise, the expectation of the makespan would be infinite!).

1.2.2 Fail-stop failures

By contrast with soft errors, relatively few published works have studied fail-stop failures in the context of workflow applications. Fail-stop errors have far more drastic consequences than soft errors as they induce the loss of all data present in memory. Therefore they require different solutions.

Consider first a workflow that consists of a linear chain of tasks. The problem of finding the optimal checkpoint strategy, i.e., of determining which tasks to checkpoint, in order to minimize the expected execution time, has been solved by Toueg and Babaoglu [102] using a dynamic programming algorithm. Note that the tasks can themselves be parallel, but the execution flow is sequential, which dramatically limits the amount of re-execution in case of a failure. The algorithm of [102] was later extended in [11] to cope with both fail-stop and silent errors simultaneously.

Consider now a general workflow comprised of parallel tasks that each executes on the whole platform. Therefore, the workflow execution is linearized, and in essence reduces to a chain of macro-tasks executing on a single macro-processor, whose speed is the aggregate speed of the available processors, and whose failure rate is proportional to the number of available processors. Checkpoints can then be placed after some tasks. However, because the original workflow is not a chain, it is more complicated to keep track of live output data, and the problem of placing checkpoints is NP-complete for simple join graphs [6]. To circumvent this problem, when checkpointing a task, one can decide to checkpoint not only the task's own output data, but also all the live data that will be needed later on in the workflow. This is the main idea of the algorithm proposed in Section 2.4.

Finally, consider a general workflow whose tasks do not span the whole platform when executing. Existing work in this most general context diverges from ours as follows: either there is a limit to the number of failures that an execution can cope with, or the optimization objective is reliability, meaning that application execution can fail, or the method design is based on specific domain knowledge.

Limiting the number of possible failures renders the problem more tractable (and is done also in the context of silent errors [55]). For instance, Wang et al. [107] present a replication-based approach, called Imitator, which is only guaranteed to succeed when no more than k fail-stop failures occur during a given execution of a DAG (which is executed repeatedly), assuming that there are $k + 1$ replicas.

In terms of works that target application reliability, is the work by Assayad et al. [4] on multi-criteria scheduling for real-time systems. They try to simultaneously minimize the application makespan and the probability that an execution succeeds. Jacques-Silva et al. describe in [61] a modeling framework for evaluating the dependability of streaming applications under faults that lead to data loss or silent data corruption. This framework is used to compare three fault tolerance techniques, including checkpointing. However, in their models, even checkpoints do not guarantee a successful application execution.

In [122], Zhu et al. proposed an automatic checkpoint algorithm on Spark to solve the long lineage problem. Because of the lazy feature of transformation operation, one could get the logical graph (lineage) before an action (submitting a job). Their solution is just tracing back the lineage, find and keep all the RDDs which are created in the job with direct parents in the previous job so they could recompute from these RDDs to get all the RDDs in this job. This method is based on specific domain knowledge. In [69], they designed a checkpointing mechanism specifically for Map tasks, which creates a checkpoint at a specific percentage of a task in progress. Checkpoints are created when the progress reaches 0.5 or 0.25 by calculation progress rate and estimated task execution time. Hwang et al. [60] presented a flexible framework for handling Grid failures. They divided workflow failure handling techniques into two different levels, namely task-level and workflow-level. Duan et al. [37] presented the Distributed workflow Enactment Engine (DEE) of the ASKALON application. Their approach focuses on application-level checkpoints which triggered by precise checkpointing event defined. In [100], they described an alternative workflow-level chekpointing scheme and every workflow node accomplishes its own local checkpoint. They employed the Petri nets to model the workflows which is a hierarchical non-DAG structure.

To the best of our knowledge, this work is the first approach (beyond application-specific solutions) that (i) does not resort to linearizing the entire workflow as a chain of (macro-)tasks; (ii) can cope with an arbitrary number of failures; (iii) always guarantees a successful application execution; and (iv) minimizes the (expectation of) the application execution time. As a result, we propose the first DAG scheduling/checkpointing algorithm that allows arbitrary workflows to execute concurrently on multiple failure-prone processors in standard task-parallel fashion.

1.2.3 Branch and bound methods

Makespan minimization is a difficult problem, and finding optimal or near-optimal solutions is computationally expensive. Since Greenberg, in his pioneering paper [43], has proposed using the branch and bound approach for scheduling DAGs, many authors have proposed methods based on meta-heuristics that prune large portions of the solution tree. The main idea is to use some (typically complicated) evaluation function to estimate the makespan that can be achieved when exploring a solution sub-tree. Task bottom-levels and critical paths are at the heart of such evaluation functions. A paper with representative state-of-the-art results is by Shahul and Sinnen [89], who use the A* meta-heuristic to optimally schedule DAGs with up to 40 tasks. Combinatorial explosion prohibits scheduling larger DAGs. This remains true with parallel implementations of branch and bound methods and these implementations do not scale well (e.g., in [57] the authors report that parallel speedup plateaus when using more than 16 processors).

A limitation of our work is that we decouple the allocation of tasks to processors and the checkpointing decisions. While the final part of the checkpointing stage is optimal, it may be possible to improve the overall approach by making allocation and checkpointing decisions simultaneously. A natural idea would be to extend previously proposed branch and bound methods by incorporating checkpoint decisions. However, there are several difficulties: (i) since the problem is more combinatorial with checkpoints, one could only hope to solve even smaller problems, perhaps with only up to around 20 tasks; and (ii) we lack an accurate evaluation function to prune solution sub-trees. Branch and bound

approaches require a lower-bound on the objective achievable by any solution in the considered branch. In our context, obtaining such a bound seems a very challenging problem as we do not even know how to estimate the expected execution time of a complete solution analytically and have to resort to Monte-Carlo simulations instead. Therefore, new breakthroughs are needed if one hopes to extend branch and bound methods to failure-aware DAG scheduling.

Chapter 2

Optimal solutions for special classes of task graphs

In this chapter, we build our first approach which is to avoid crossover dependencies by restricting the problem to a particular class of workflow DAGs: Minimal Series Parallel Graphs (M-SPGs) [104]. Despite their name, M-SPGs extend classical Series Parallel Graphs (SPGs) [15] by allowing source and sink nodes to not be merged in series composition (see Section 2.2.3 for details). It turns out that most real-world workflows, e.g., those executed today by production WMSs [1, 2, 32, 38, 109, 110], are M-SPGs. The structure of these graphs makes it possible to orchestrate the execution in fork-join fashion, by which processors compute independent sets of tasks, before joining and exchanging data with other processors. We call these independent sets of tasks *superchains*, because tasks in these sets are linearized into a chain (because they are executed by a single processor) but have forward dependencies that can "skip over" immediate successors. We remove all crossover dependencies by always checkpointing the output data of the exit tasks of a superchain, thus removing the difficulties associated with processor interference.

In this chapter we propose, to the best of our knowledge, the first scheduling/checkpointing strategy for minimizing the expected makespan of workflow applications with fail-stop failures that considers workflow structures more general than mere linear chains of tasks. More specifically, our contributions are:

- A scheduling/checkpointing strategy, PROPCKPT, for M-SPGs that improves upon both the de-facto standard CKPTALL strategy and the CKPTNONE strategy (Section 2.2). PROPCKPT avoids all crossover dependencies and relies on the two algorithms below;
- A list-scheduling algorithm, which is inspired by the "proportional mapping" approach [80], for scheduling M-SPG workflows as sets of superchains (Section 2.3);
- An algorithm, which extends the dynamic programming algorithm by Toueg and Babaoğlu [102], to checkpoint tasks in a superchain optimally (Section 2.4);
- The #P-completeness of the problem of computing the expected makespan for the CKPTNONE strategy (Section 2.5). To the best of our knowledge, the complexity of computing, or even approximating, the expected makespan for CKPTNONE was an open problem;
- Experimental evaluation with real-world Pegasus [32] workflows to quantify the performance gains afforded by our proposed approach in practice (Section 2.6).

In addition to the above sections, Section 2.1 details an example with crossover dependencies, and Section 2.7 provides concluding remarks and highlights directions for future work.

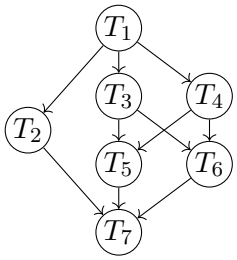


Figure 2.1: Example task graph.

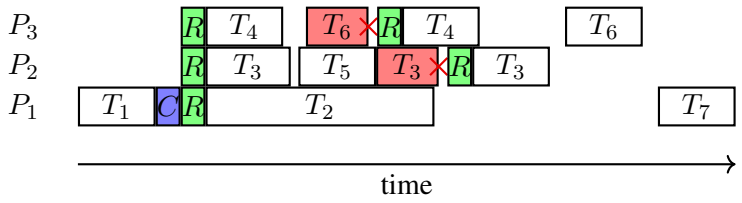


Figure 2.2: Example schedule of the workflow in Figure 2.1 on 3 processors. Processor P_3 fails while executing task T_6 and has to start all its tasks again. However, processor P_2 then fails and the result of task T_3 is lost, delaying the re-execution of task T_6 until task T_3 is re-executed on processor P_2.

2.1 Example

Consider the workflow in Figure 2.1, which comprises 7 tasks, T_i, $1 \leq i \leq 7$. The execution of this workflow on 3 processors, P_i, $1 \leq i \leq 3$, for a given schedule is shown in Figure 2.2. Two failures occur during the execution, first on P_3 and then on P_2 (shown as red X's on the figure). In this example, T_1 is checkpointed. The checkpoint overhead for saving T_1's output to stable storage occurs immediately after T_1 completes (shown as C on the figure). All successors of T_1 must then "recover" from that checkpoint to begin execution, which also has some overhead (shown as R's on the figure before the execution of T_2, T_3 and T_4). No other task besides T_1 is checkpointed in this example. As a result, some direct (i.e., not via stable storage) communication is required between some tasks. Figure 2.2 shows delays due to these communications. For instance, the first execution of T_6 on P_3 does not start immediately after T_3 completes but only after a delay, which corresponds to the time for P_2 to send T_3's output to P_3.

For non-checkpointed tasks and their successors, one must define precisely how inter-processor communications take place, i.e., when the data is transferred and for how long it is stored at the sender and the receiver. Recall that when a failure occurs on a processor, the whole content of that processor's memory is lost. As a result, the way in which inter-processor communications take place can impact the failure recovery procedure. For instance, in Figure 2.2, a failure strikes P_2 after the completion of T_5 but before T_7 begins executing on P_1. If the output data of T_5 is sent to P_1 as soon as T_5 completes, there is no need to re-execute T_5 for executing T_7. On the contrary, if this output is sent as late as possible (i.e., so that it is received just before the execution of T_7 begins), then T_5 will need to be re-executed because its output will have been lost on P_2 due to the failure. For the sake of the example in Figure 2.2, we have chosen the former option. More formally, consider a non-checkpointed task T, executed on a processor P, that produces data that is needed by task T' on a different processor P' (i.e., a crossover dependency as described in Section 1.1). Then, the data is transferred from P to P' immediately after T completes. This data is deleted from memory on P as soon as the data transfer has completed, and deleted in memory on P' after T' completes. If T and T' are scheduled on the same processor P, then the output data of T is in memory of P from the completion of T until the completion of T'.

There are three phases in the schedule in Figure 2.2: (i) execution until the first failure; (ii) recovery and execution until the second failure; and (iii) recovery and termination of the execution. The result of the execution of T_1 on processor P_1 is saved to stable storage because T_1 is checkpointed. Therefore, T_1 will never need to be re-executed once it has executed successfully. The three successors of T_1, namely T_2, T_3 and T_4, start their executions after reading the output data of T_1 from stable storage. Upon completion, the results of T_3 and T_4 are transmitted to their successors on other processors immediately.

The first failure interrupts the execution of T_6 on processor P_3. Due to this failure, P_3 loses the output data of T_3 and T_4, which are required to execute T_6. Hence, both T_3 and T_4 must be restarted on processors P_2 and P_3, respectively. The result of T_1 is recovered from stable storage in order to allow the execution of T_4 on P_3. Processor P_2 still has the output of T_1 in memory, and thus does not perform this recovery. However, at the time of the failure, T_5 is running on P_2, and then the re-execution of T_3 can only start once T_5 completes. This is because we have made the common implicit assumption that tasks are non-preemptible.

The second failure interrupts the re-execution of task T_3 on P_2. P_2 then re-executes T_3, which requires recovering the output of T_1 from stable storage. Once T_3 completes on P_2, then P_3 can execute T_6. Finally, T_7 is executed on processor P_1 after T_6 completes.

This example highlights the difficulties caused by crossover dependencies, here from T_3 to T_6 (where a failure on P_3 causes a re-execution on P_2) and from task T_5 to task T_7 (for which a failure may cause a re-execution depending on assumptions on when inter-processor communications are performed). The main observation is that, with crossover dependencies, a failure on a processor can cause task re-executions on other processors. These re-executions are themselves subject to failures, and these failures can also cause re-executions on yet other processors. As a result, a failure on one processor can "ripple" through all processors. As a result, estimating (and thus minimizing) the expected makespan is hopelessly combinatorial. None of the known methods designed to approximate the expected makespan of DAGs with probabilistic task durations can be applied. To the best of our knowledge, the only option would be to use discrete event simulation and hope to estimate the expected makespan as an average over a large number of trials with randomly injected failures.

In the example in Figure 2.2, avoiding all crossover dependencies would require checkpointing four additional tasks: T_3, T_4, T_5 and T_6. With these additional checkpoints, a task re-execution on a processor only happens if a failure occurs on that processor. This avoids the failure rippling effect described above. Without crossover dependencies, we are back to a situation where failures are contained to individual processors, just as when all tasks are checkpointed in the CKPTALL approach.

Given the above, in this work we avoid crossover dependencies altogether. Once these dependencies are eliminated, our approach views all the tasks executed by a same processor in between two checkpoints as a single (larger) task. In other terms, we logically coalesce a group of consecutive not-checkpointed tasks followed by a checkpointed task into a single checkpointed task. With this logical coalescing, we can evaluate the expected makespan as that of a DAG with probabilistic task durations, for which approximation methods exist. In summary, prohibiting crossover dependencies not only contains the impact of failures, but it also enables us to estimate the expected makespan of a solution for M-SPGS using the existing approximation methods.

2.2 Preliminaries

In this section, we first detail the execution model (DAGs) and the fault-tolerance model for failures and checkpoints that concern both Chapter 2 and Chapter 3. Then we define M-SPGS, a special class of workflow DAGs that we consider in this chapter. Next, we provide an overview of our proposed approach, including how we schedule and checkpoint tasks. Finally, we briefly review methods to compute the expected longest path in a DAG with probabilistic task durations.

2.2.1 Execution model

The execution model for a task workflow on a homogeneous system is represented as a Directed Acyclic Graph (DAG), $G = (V, E)$, where V is the set of nodes corresponding to the tasks, and E is the set of

edges corresponding to the dependencies between tasks. In a DAG, a node without any predecessor is called an *entry* node, while a node without any successor is an *exit* node. For a task T in G, Pred(T) and Succ(T) represent the set of its immediate predecessors and successors respectively. We say that a task T is *ready* if either it does not have any predecessor in the dependency graph, or if all its predecessors have been executed.

In this model, Each task $T_i \in V$ has a weight (in flops), w_i, i.e., its execution time in a failure-free execution. Each dependency $(T_i, T_j) \in E$ is weighted by the size of communicated data (in bytes). We could compute the cost, c_{ij}, to store/read the data produced by task T_i and needed by task T_j, onto/from stable storage, by dividing the size of the file in bytes by the bandwidth of the stable storage in byte/sec. In some real-world instances from [78], a task may generate multiple files for different successors or generate the same file for more than one successor task. We always start executing the task after checkpointing the crossover dependency, to make sure that all files needed are available in the stable storage in case of failure.

Data transfer between two processors (i.e., a *crossover dependency*) consists in writing and reading files from the stable storage in this chapter. While in the general case in the following chapter, direct communications between processors take place through a communication network which is much faster than writing/reading from the stable storage. Before the execution of T_j on processor P_k, all input files needed by T_j must be present in the local memory of P_k. Suppose that T_j needs a file from T_i executed on processor P_l, it will first try to retrieve the file from the local memory of P_l. If the file is absent from local memory of P_l, then it must be *read* from the stable storage. The start time of T_j is $\max\{t_{c_{ij}}, (t_{avail_k} + t_c)\}$, where $t_{c_{ij}}$ is the finish time of checkpointing the file transferred from T_i to T_j, t_{avail_k} donates the available time on processor P_k, and t_c is the communication time.

2.2.2 Fault-tolerance model

In this part, we consider failure-prone processors that stop their execution once a failure occurs (i.e., we have fail-stop errors). When a fail-stop error strikes, the execution stops, the entire content of the processor memory is lost, and computation has to be started from scratch, either on the same processor after a reboot, or on a spare processor (e.g., taken from a pool of spare processors specifically requested by the job submitter, taken from a pool of spare processors maintained by the resource management infrastructure). Each processor is a processing element that is subject to its own individual failures. Failures can strike a processor at any time, during either task execution or waiting time (e.g., the power supply may fail).

Consider a single task T, with weight w, scheduled on such a processor, and whose input is stored on stable storage. It takes a time r to read that input data from stable storage, either for its first execution or after a failure. The total execution time W of T is a random variable, because several execution attempts may be needed before the task succeeds. We assume that failures are i.i.d. (independent and identically distributed) across the processors and that the failure inter-arrival times at each processor is Exponentially distributed with Mean Time Between Failures (MTBF) $\mu = 1/\lambda$. Let $\lambda \ll 1$ be the Exponential failure rate of the processor. With probability $e^{-\lambda(r+w)}$, no failure occurs, and W is equal to $r + w$. With probability $(1 - e^{-\lambda(r+w)})$, a failure occurs. For Exponentially distributed failures, the expected time to failure, knowing that a failure occurs during the task execution (i.e., in the next $r + w$ seconds), is $1/\lambda - (r + w)/(e^{\lambda(r+w)} - 1)$ [56]. After this failure, there is a downtime d, which is (an upper bound of) the time to reboot the processor or migrate to a spare. Then we start the execution again, first with the recovery r, the time to read data from stable storage, and then the work w. With a general model where an unbounded number of failures can occur during recovery and work, the expected time W to execute task T is given by $W = \left(\frac{1}{\lambda} + d\right)\left(e^{\lambda(r+w)} - 1\right)$ [56]. Now if the output data of task T is

checkpointed, with a time c to write it onto stable storage, the total time becomes:

$$W = \left(\frac{1}{\lambda} + d\right)\left(e^{\lambda(r+w+c)} - 1\right) . \tag{2.1}$$

Equation 2.1 assumes that failures can also occur during checkpoints, which is the most general model for failures. In the case of a sequence of non-checkpointed tasks to be executed on a processor P, the output data of each task resides in the memory of P for re-use by subsequent tasks. When a failure strikes P, the entire memory content is lost and the whole task sequence must be re-executed from scratch. As a side note, Equation 2.1 is an upper bound, as we assume that all input files to task T are stored on stable storage, the cost r is paid for starting the first execution as well as for the recovery after a failure. In practice, some input files may already be present in local memory (e.g., generated by predecessors of T that scheduled on the same processor) or may be transferred from other processors through communication instead of reading from stable storage. As a result, when no failure strikes, to start the first execution of T, one could pay less than r. Because the case is different from task to task, and we have no simple mean to know whether some failures had previously struck, we have to resort to this upper bound. [1]

2.2.3 Minimal Series Parallel Graphs (M-SPG)

In this chapter, we consider computational workflows structured as Minimal Series Parallel Graphs (M-SPGs) [104], which (despite their name) are generalizations of standards SPGs [15]. An M-SPG is a restricted type of graph $G = (V, E)$, which is defined recursively based on two operators, $\overrightarrow{;}$ and $||$, defined as follows:

• The *serial composition* operator, $\overrightarrow{;}$, takes two graphs as input and adds dependencies from all sinks of the first graph to all sources of the second graph. Formally, given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, $G_1 \overrightarrow{;} G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup (sk_1 \times sc_2))$, where sk_1 is the set of sinks of G_1 and sc_2 the set of sources of G_2. This is similar to the serial composition of SPGs, but without merging the sink of the first graph to the source of the second, and extending the construct to multiple sources and sinks.

• The *parallel composition* operator, $||$, simply makes the union of two graphs. Formally, given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, $G_1||G_2 = (V_1 \cup V_2, E_1 \cup E_2)$. This is similar to the parallel composition of SPGs, but without merging sources and sinks.

Given the above operators, an M-SPG is then defined recursively as follows:

• A *chain* $g_1 \overrightarrow{;} \ldots \overrightarrow{;} g_n$, where each g_i is an atomic task;

• A serial composition $G_1 \overrightarrow{;} \ldots \overrightarrow{;} G_n$, where each G_i is an M-SPG; or

• A parallel composition $G_1|| \ldots ||G_n$, where each G_i is an M-SPG.

Figure 2.3 shows example M-SPG structures. Due to the above definition supporting multiple sources and sinks, and not merging sources and sinks, M-SPGs naturally support fork, join (and therefore fork-join), and bipartite structures. It turns out that these structures are common in production workflow applications. For instance, most workflows from the Pegasus benchmark suite [32, 78] are M-SPGs. Overall, M-SPGs exhibit the recursive structure of SPGs (which is key to developing tractable scheduling/checkpointing solutions), but are more general, and as a result maps directly to most production workflow applications. In particular, M-SPGs can model communication patterns that cannot be modeled with SPGs (this is the case of the bipartite structure shown in Figure 2.3(c)).

[1] If we assume no read time for the first execution (e.g., all input files exist in local memory), then the value of W becomes $\left(\frac{1}{\lambda} + d\right) e^{\lambda r}\left(e^{\lambda(w+c)} - 1\right)$.

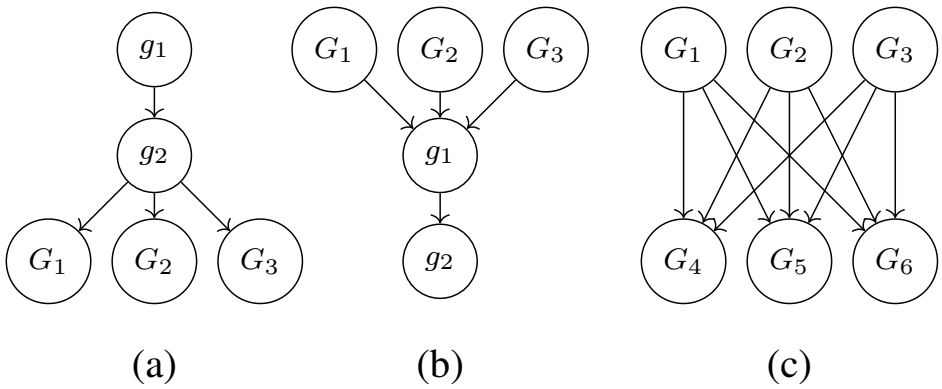


Figure 2.3: Example M-SPG structures (g_1 and g_2 are atomic tasks whereas G_1 to G_6 are M-SPGs): (a) fork: $(g_1 \vec{;} g_2) \vec{;} (G_1||G_2||G_3)$; (b) join: $(G_1||G_2||G_3) \vec{;} (g_1 \vec{;} g_2)$; (c) bipartite: $(G_1||G_2||G_3) \vec{;} (G_4||G_5||G_6)$.

2.2.4 Problem description and proposed approach

As outlined in Section 1.1, our objective is to not checkpoint all output data, so as to save on checkpointing overhead and thus reduce the expected overall execution time, or makespan. The expected makespan includes the checkpointing and recovery overheads. This leads naturally to the following optimization problem: given an M-SPG to execute on processors that experience failures with a given Exponential rate, compute a schedule that does not involve direct inter-processor communications and that minimizes the expected makespan. The schedule must specify which processor executes which tasks, when each task begins execution, and which output data is checkpointed and when.

Our PROPCKPT approach computes a schedule that allocates tasks to processors, and that avoids direct inter-processor communications by checkpointing particular tasks so as to remove all crossover dependencies (See Section 2.1). It then optimally determines which additional tasks should be checkpointed so as to minimize the expected makespan.

Consider an M-SPG, G. Without loss of generality, $G = C \vec{;} (G_1||\ldots||G_n) \vec{;} G_{n+1}$, where C is a chain and $G_1, \ldots, G_n, G_{n+1}$ are M-SPG graphs, with some of these graphs possibly empty graphs. The schedule for G is the temporal concatenation of the schedule for C, the schedule for $G_1||\ldots||G_n$, and the schedule for G_{n+1}. A chain is always scheduled on a single processor, with all its tasks executed in sequence on that processor. When scheduling a parallel composition of M-SPGs, we use the following polynomial-time list-scheduling approach, inspired by the "proportional mapping" heuristic [80]. Given an available number of processors, we allocate to each parallel component G_i an integral fraction of the processors in proportion to the sum of the task weights in G_i (the overhead of reading/writing data to stable storage is ignored in this phase of our approach). In other terms, we allocate more processors to more costly graphs. We apply this process recursively, each time scheduling a sub-M-SPG on some number of processors. Eventually, each sub-M-SPG is scheduled on a single processor, either because it is a chain or because it is allocated to a single processor. In this case, all atomic tasks in the M-SPG are linearized based on a topological order induced by task dependencies and scheduled sequentially on the processor. This algorithm is described in Section 2.3.

Each time a sub-M-SPG is scheduled on a single processor, we call the set of its atomic tasks a *superchain*, because the tasks are executed sequentially even though the graph may not be a chain. We call the *entry tasks*, resp. *exit tasks*, of a superchain the tasks in the superchain that have predecessors, resp. successors, outside the superchain. Due to the recursive structure of an M-SPG, all predecessors of the entry tasks in a superchain are themselves exit tasks in other superchains. Similarly, all successors

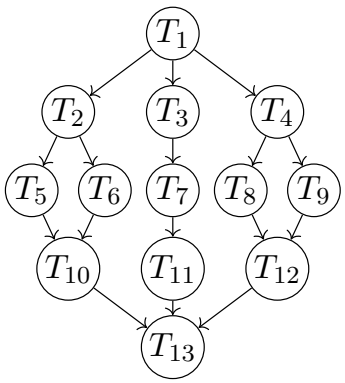


Figure 2.4: Example M-SPG.

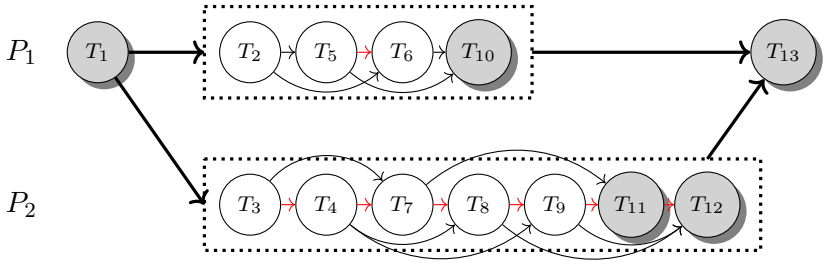


Figure 2.5: Mapping the M-SPG of Figure 2.4 onto two processors. The two superchains are shown inside boxes, with all internal and external dependencies from the original graph (red edges result from the linearization). T_{10} is the only exit task of the top superchain while T_{11} and T_{12} are the two exit tasks of the bottom superchain. A checkpoint is performed to save the output of each shadowed task.

of the exit tasks in a superchain are themselves entry tasks in other superchains. This has two important consequences:

• The workflow is an "M-SPG of superchains";

• Checkpointing the output data of all exit tasks of a superchain means that this superchain never needs to be re-executed. In this case, we say that "the superchain is checkpointed".

A natural strategy is then to checkpoint all superchains, which avoids all crossover dependencies. More specifically, a systematic checkpoint that saves the output data of all exit tasks of a superchain is performed after the last task of that superchain completes. This checkpoint strategy is detailed in Section 2.4.1. Figure 2.5 shows an example of a schedule obtained on two processors for the M-SPG in Figure 2.4. A set of tasks is linearized on each processor (additional dependencies are added to enforce sequential execution of tasks on a single processor). Five checkpoints are taken: after the executions of T_1, T_{10}, T_{11}, T_{12} and T_{13}. This guarantees that failures are contained: Once T_{13} begins executing, a failure on P_2 has no effect and a failure on P_1 is handled by immediately re-starting T_{13} after a downtime and recovery.

The above approach produces a solution with the lowest number of checkpoints necessary to avoid crossover dependencies. To evaluate the expected makespan, one can then coalesce all tasks in a superchain into a single checkpointed task, leading to an M-SPG in which all tasks are checkpointed. In our example, the four tasks of the top superchain would be coalesced into one checkpointed task, and so would the seven tasks of the bottom superchain. One can then estimate the expected makespan using

known algorithms for DAGs with probabilistic task durations (see Section 2.2.5). While this approach avoids all crossover dependencies, and thus makes sure that failures are contained, its expected makespan may be far from optimal because too few tasks are checkpointed. Depending on the parallelism of the M-SPG and the total number of available processors, superchains may contain large numbers of tasks. If only the output data of exit tasks are checkpointed, then the expected execution time of a superchain can be large due to many re-executions from scratch. One should then checkpoint additional output data throughout the execution of the superchain. To this end, we propose a polynomial-time dynamic programming algorithm that extends the approach of Toueg and Babaoğlu [102] to determine the optimal set of output data to checkpoint. This algorithm is described in Section 2.4.2. Once these additional checkpoints are determined, thereby creating sequences of tasks followed by a checkpoint, we logically coalesce these sequences into a single task. Again, this is so that we can use known algorithms for estimate the expected makespan of DAGs with probabilistic task durations.

2.2.5 Evaluation of expected makespan

As discussed in Section 1.1, computing the expected makespan for a solution with the CKPTALL strategy (tasks being already assigned to processors and all checkpointed) amounts to computing the expected longest graph of a DAG with probabilistic task durations. Recall that, once scheduled, the original workflow graph is augmented with extra dependencies to enforce serial executions of tasks at each processor.

In the DAG, task weights are random variables whose expectation is given by Equation 2.1. The CDF of such a random variable is complicated, because one has to account for the possibility of an arbitrary number of failures occurring at arbitrary instants. To the best of our knowledge, there is no closed-form for this CDF.

Computing the expected longest path is #P-complete, even if one considers that the execution time of a task is a discrete random variable that can take only 2 values [46]. However, basic probability theory tells us how to compute the probability distribution of the sum of two independent random variables (by a convolution) and of the maximum of two independent random variables (by taking the product of their cumulative density functions). As a result, one can compute the makespan distribution and its expected value if the DAG is a SPG (or an M-SPG), due to its recursive structure [19, 74]. However, the makespan may take an exponential number of values, which makes its direct evaluation inefficient. With only 2 values, the problem of computing the expected makespan remains NP-complete, but in the weak sense, and admits a pseudo-polynomial solution [74]. With complicated distributions for task weights as discussed above, the evaluation becomes intractable, and one has to resort to approximations.

Several approximation methods have been proposed, including approximating general graphs by series-parallel graphs [19, 74], approximating task weight distributions by Normal distributions [19, 88], or approximating the length of the longest paths [25]. Rather than using these approaches, which have various levels of accuracy depending of DAG structure, we use the classical Monte Carlo simulation approach [73, 93] with very large numbers of trials. Each trial consists in sampling the weight of each task in the DAG from its distribution. This method is compute-intensive but provides an accurate way to compare different scheduling/checkpointing strategies fairly (more accurately than using the aforementioned approximation methods).

2.3 Scheduling M-SPGs

In this section, we describe the list-scheduling algorithm of our PROPCKPT approach, by which we assign sub-graphs of the workflow DAG to processors. Our algorithm decides how many processors

should be allocated to parallel sub-graphs. It is recursive, so as to follow the recursive M-SPG structure, and produces a schedule of superchains, as explained in Section 2.2.4. It adapts the principle of "proportional mapping" heuristic [80] to M-SPGs. Pseudo-code is given in Algorithm 1.

Procedure ALLOCATE schedules an M-SPG G, which comprises sequential atomic tasks, onto a finite set $\mathcal{P}$ of processors. It returns immediately if $G = \emptyset$ (Line 2), otherwise it decomposes G into the sequential composition of a chain, C, a parallel composition, $G_1||\ldots||G_n$, and an M-SPG, G_{n+1} (Line 4). Note that several such decompositions exist and some of them lead to infinite recursions. This is the case when the chain is empty and a single graph is non-empty among $\{G_1, \ldots, G_{n+1}\}$). For instance, the graph G could be decomposed such that $C = G_1 = \ldots = G_n = \emptyset$ and $G_{n+1} = G$, or $C = G_2 = \ldots = G_{n+1} = \emptyset$ and $G_1 = G$. Our algorithm avoids these superfluous decompositions and make sure that C is the longest possible chain. It then schedules the three components in sequence. To do so, it relies on two helper procedures: the ONONEPROCESSOR procedure, which schedules tasks on a single processor, and the PROPMAP procedure, when more processors are available. ALLOCATE calls ONONEPROCESSOR to schedule C (Line 5) and to schedule $G_1||\ldots||G_n$ if a single processor is available (Line 7). If $|\mathcal{P}| > 1$, then ALLOCATE calls the second helper procedure, PROPMAP (Line 9). This procedure takes in a set of n M-SPGs and a number of processors, p, and returns a list of M-SPGs and a list of processor counts. ALLOCATE then simply recursively schedules the i-th returned M-SPG onto a partition of the platform that contains the i-th processor count (Line 10-13). Finally, ALLOCATE is called recursively to schedule G_{n+1} (Line 14).

The PROPMAP procedure is the core of our scheduling algorithm. Let $k = \min(n, p)$ be the number of returned M-SPGs and processor counts (Line 16). Initially, the k M-SPGs are set to empty graphs (Line 17), and the k processor counts are set to 1 (Line 18). Array W contains the weight of each returned M-SPGs, initially all zeros (Line 19). Then, input M-SPGs are sorted by non-increasing weight, the weight of an M-SPG being the sum of the weights of all its atomic tasks (Line 20). Two cases are then handled. If $n \geq p$, PROPMAP iteratively merges each G_i with the output M-SPG that has the lowest weight so as to obtain a total of p non-empty output M-SPGs (Line 22-25). The processor counts remain set to 1 for each output M-SPG. If instead $n < p$, then there is a surplus of processors. PROPMAP first assigns each input G_i to one output M-SPG (Line 27-29). The $p-n$ extra processors are then allocated iteratively to the output M-SPG with the largest weight (Line 30-35). Finally, PROPMAP returns the lists of output M-SPGs and of processor counts.

The ONONEPROCESSOR procedure (Line 37-40) takes as input an M-SPG and a processor, performs a topological sort of the M-SPG's atomic tasks, and then schedules these tasks, which constitute a superchain, in sequence onto the processor. After assigning all sub-graphs of G onto processors, we complete our PROPCKPT approach by calling the CHECKPOINT procedure to decide which tasks from each superchain L to checkpoint (Line 41-45), as described in Section 2.4.

2.4 Placing checkpoints in superchains

In this section, we describe our approach for deciding after which tasks in a superchain output data must be checkpointed. We first describe existing results for simple chains and explain how the problem is more difficult in the case of superchains. We then describe an optimal dynamic programming algorithm for superchains.

2.4.1 From chains to superchains

Toueg and Babaoğlu [102] have proposed an optimal dynamic programming algorithm to decide which tasks to checkpoint in a linear chain of tasks. For a linear chain, when a failure occurs during the

Algorithm 1: Algorithm PROPCKPT

```
1  Procedure ALLOCATE (G, P):
2      if G = ∅ then
3          return
4      C →; (G_1 || ... || G_n) →; G_{n+1} ← G
5      L ← ONONEPROCESSOR (C, P[0])
6      if (|P| = 1) then
7          L ← L ∪ ONONEPROCESSOR (G_1 || ... || G_n, P[0])
8      else
9          (Graphs, Counts) ← PROPMAP (G_1, ..., G_n, |P|)
10         i ← 0
11         for each graph, count in Graphs, Counts do
12             ALLOCATE (graph, {P[i], ..., P[i + count − 1]})
13             i ← i + count
14     return L ∪ ALLOCATE (G_{n+1}, P)
15 Procedure PROPMAP (G_1, ..., G_n, p):
16     k ← min(n, p)
17     Graphs ← [∅, ..., ∅] (k elements)
18     procNums ← [1, ..., 1] (k elements)
19     W ← [0, ..., 0] (k elements)
20     Sort [G_1, ..., G_n] by non-increasing total weight
21     if n ≥ p then
22         for i = 1 ... n do
23             j ← arg min_{1≤q≤p} (W[q])
24             W[j] ← W[j] + weight(G_i)
25             Graphs[j] ← Graphs[j] || G_i
26     else
27         for i = 1 ... n in G_i do
28             Graphs[i] ← G_i
29             W[i] ← weight(G_i)
30         ρ ← p − n
31         while ρ ≠ 0 do
32             j ← arg max_{1≤q≤n} (W[q])
33             procNums[j] ← procNums[j] + 1
34             W[j] ← W[j] × (1 − 1/procNums[j])
35             ρ ← ρ − 1
36     return Graphs, procNums
37 Procedure ONONEPROCESSOR (G, P):
38     L ← topological_sort(G)
39     MAP (L, P)                          // Schedule tasks serially on one processor
40     return {L}
41 Procedure PROPCKPT (G, P):
42     L ← ALLOCATE (G, P)
43     for L ∈ L do
44         CHECKPOINT (L)                  // Decide which tasks to checkpoint
45     return
```

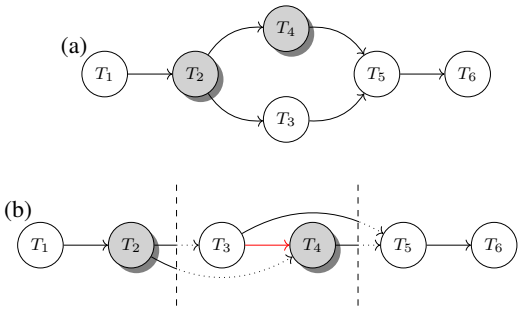


Figure 2.6: (a) Example M-SPG in which checkpointed tasks (T_2 and T_4) are shadowed. (b) Linearization of the M-SPG on a single processor. The dependency from T_3 to T_4, in red, results from the linearization. Vertical dashed lines correspond to checkpoints (after T_2 and T_4). Dotted lines correspond to dependencies from tasks that have been checkpointed.

execution of a task T, one has to recover from the latest checkpoint and re-execute all non-checkpointed ancestors of T. In this work, we target M-SPG (sub-)graphs that are linearized on a single processor. As a result, recovery from failure is more complex than in the case of a linear chain. Consider a failure during the execution of a task T. For T to be re-executed, all its input data must be available in memory. Therefore, for each reverse path in the graph from T back to entry tasks of the superchain, one must recover from the latest checkpoint, and then recover by re-executing all non-checkpointed ancestors of T along each reverse path. Consider the M-SPG in Figure 2.6(a), and its linearization on a single processor in Figure 2.6(b). Let us assume that tasks T_2 and T_4 are checkpointed (shadowed in the figures). According to the standard definition of checkpoints, the checkpoint of T_2 includes both its output for T_3 and its output for T_4, while the checkpoint of T_4 includes only its output for T_5.

Let us now consider a single failure that occurs during the execution of T_5. To re-execute T_5, one needs to recover from the checkpointed output of T_4. But one also needs to re-execute T_3, which was not checkpointed, since the output of T_3 is needed for executing T_5. To re-execute T_3, one needs to recover from the checkpoint of T_2. This sequence of recoveries and re-executions must be re-attempted until T_5 executes successfully. As a result, the problem of deciding which tasks to checkpoint to minimize expected makespan cannot be solved by the simple linear chain algorithm in [102], which relies on a single recovery from the latest checkpoint followed by the re-execution of all tasks executed since that checkpoint.

We thus propose an alternative approach by which a checkpoint, which takes place after the execution of a task (named the *task checkpoint*), saves not only the output data from that task, but also all non-checkpointed output data from previously executed tasks. In other words, during a checkpoint, all non-checkpointed data that is available in the processor's memory is saved to stable storage. Each such data is input to a task that is yet to be executed. The rationale is that this data needs to be saved so that after a failure execution can be restarted from the checkpointed task. This is shown in Figure 2.6, where checkpoint times are depicted as vertical dashed lines, after each execution of a checkpointed task (in this case T_2 and T_4). "Taking a task checkpoint" means saving to stable storage all output data of previously executed but un-checkpointed tasks. Visually, this corresponds to solid dependency edges that cross the checkpoint time, as shown in Figure 2.6. With this extended definition of checkpoints, the checkpoint after T_4 now includes the output data of T_3 for T_5, in addition to the output of T_4 for T_5. This approach allows the algorithm in [102] to be extended to the case of superchain as described in the next section.

2.4.2 Checkpointing algorithm

To answer the question of when to take checkpoints throughout the execution of a superchain on a processor, we propose an $O(n^2)$ dynamic programming algorithm. For each sequence of tasks allocated to a processor, the algorithm finds the optimal set of tasks after which output data must be checkpointed in order to minimize its expected completion time. For each task of a sequence, it determines the position of the last checkpoint that optimizes the expected completion time. The set of tasks after which a checkpoint is taken can then be obtained by backtracking from the last task of the superchain.

Let us consider a superchain that contains tasks $T_a, \ldots, T_b$ (we assume that tasks $T_1, \ldots, T_n$ are numbered according to a topological sort in such a way that tasks from any superchain have contiguous indices). Without loss of generality let us assume that T_j executes immediately before T_{j+1}, $j = a, \ldots, b-1$ and that T_a starts as soon as the necessary input data is read from stable storage.

Our approach always takes a checkpoint after T_b completes to avoid crossover dependencies (see Section 2.2.4), thus ensuring that all output data from all exit tasks of the superchain are checkpointed. Let $\mathcal{E}Time(j)$ be the optimal expected time to successfully execute tasks $T_a, \ldots, T_j$, when a checkpoint is taken immediately after T_j completes (with possibly earlier checkpoints). Our goal is to minimize $\mathcal{E}Time(b)$. To compute $\mathcal{E}Time(j)$, we formulate the following dynamic program by trying all possible locations for the last checkpoint before T_j:

$$\mathcal{E}Time(j) = \min\left(T(a,j), \min_{a \leq i < j}\{\mathcal{E}Time(i) + T(i+1,j)\}\right),$$

where $T(i+1,j)$ is the expected time to successfully execute tasks T_{i+1} to T_j, provided that a checkpoint occurs after task T_j completes and the previous checkpoint occurred before task T_{i+1} starts. This account for the time to read the input data, execute the tasks and perform the checkpoint. As there is no checkpoint between tasks T_{i+1} and T_j, all intermediate data are kept in memory and retrieved instantly. This reduces the checkpoint overhead compared to CKPTALL.

From Equation 2.1, the expected time needed to execute tasks T_i to T_j for each (i,j) pair with $i \leq j$ is given by

$$T(i,j) = \left(\frac{1}{\lambda} + d\right)\left(e^{\lambda(R_i^j + W_i^j + C_i^j)} - 1\right), \tag{2.2}$$

where λ is the processor's exponential failure rate, R_i^j is the time necessary to read from stable storage all data produced by tasks $T_1, \ldots, T_{i-1}$ and needed by tasks $T_i, \ldots, T_j$, $W_i^j = w_i + \ldots + w_j$ is the time to execute tasks T_i to T_j when no failures occur, C_i^j is the time taken to checkpoint the input data of $T_{j+1}, \ldots, T_n$ that is produced by $T_i, \ldots, T_j$ (i.e., the non-checkpointed predecessors of $T_{j+1}, \ldots, T_n$ in $T_i, \ldots, T_j$), and d is the downtime. Formally, $R_i^j = \sum_{k=i}^{j} \sum_{T_l \in \text{Pred}(T_k) \setminus \{T_i, \ldots, T_j\}} c_{lk}$ and $C_i^j = \sum_{k=i}^{j} \sum_{T_l \in \text{Succ}(T_k) \setminus \{T_i, \ldots, T_j\}} c_{kl}$ where c_{kl} is the cost to read or write the data produced by T_k and needed by T_l, $\text{Pred}(T_k)$ is the set of predecessors of T_k and $\text{Succ}(T_k)$ is the set of successors of T_k. Note that the data that is read (during R_i^j) may be produced by exit tasks of previous superchains and that the data that is saved (during C_i^j) may be needed by entry tasks in next superchains. In particular, C_i^j is greater than or equal to the time to checkpoint all output data of T_j.

The pseudo-code for this dynamic programming solution is given in Algorithm 2. The computation of $\mathcal{E}Time(j)$ takes $O(n)$ time, as it depends on at most j other entries. The computation of $T(i,j)$ for all (i,j) pairs with $i \leq j$ takes $O(n^2)$ time. Therefore, the overall complexity is $O(n^2)$.

Algorithm 2: CHECKPOINT

```
1  Procedure CHECKPOINT (T_a, ..., T_b):
2      last_ckpt ← [0, ..., 0] (b − a + 1 elements)
3      for j = a ... b do
4          ETime(j) ← T(a, j)
5          last_ckpt[j] ← 0
6          for i = a ... j − 1 do
7              temp ← ETime(i) + T(i + 1, j)
8              if temp < ETime(j) then
9                  ETime(j) ← temp
10                 last_ckpt[j] ← i
11     Ckpts ← ∅                                   // List of tasks to checkpoint
12     while b ≠ a                                 // Backtracking
13     do
14         Ckpts ← Ckpts ∪ {T_b}                   // Checkpoint after task T_b
15         b ← last_ckpt[b]
16     return Ckpts
```

2.4.3 Technical remarks

Remark #1 – Our model assumes that faults may occur while reading or writing data to stable storage. We could also use the simpler assumption that faults only occur during computations, as done in many previous works, by replacing Equation 2.2 by $(\frac{1}{\lambda} + d)(e^{\lambda W_i^j} - 1) + R_i^j + C_i^j$.

Remark #2 – It may appear wasteful to read files from stable storage that were just written by the same processor and that may still be in memory or accessible locally on disk. An alternative strategy would be to assume that the processor still has access to each data item that it has computed before (until a failure strikes). The initial read time would then be reduced to $R_i^j - \sum_{k=i}^{j} \sum_{T_l \in \text{Pred}(T_k) \cap \text{Alloc}(T_i) \setminus \{T_i, \ldots, T_j\}} c_{lk}$ where $\text{Alloc}(T_i)$ is the set of tasks allocated to the same processor as T_i (i.e., we no longer include the task predecessors that were on the same processor). However, the recovery cost should also include the time to read all these data back from stable storage in case of failure. This cost would be $R_i^j + \sum_{k=1, T_k \in \text{Alloc}(T_i)}^{i-1} \sum_{T_l \in \text{Succ}(T_k) \cap \text{Alloc}(T_i) \setminus \{T_1, \ldots, T_j\}} c_{kl}$. Note that in case of multiple failures on the same processor, data may be read back from stable storage more than once, which is also wasteful (it would be more efficient to recover data whenever they are needed instead), but this overestimation of the recovery cost is necessary to apply our dynamic programming approach. For low failure probability, it may thus be advantageous to use this alternative strategy, because the higher recovery cost is offset by the lower initial cost. We did not explore this option further, but our method can be easily extended to encompass it.

Remark #3 – Algorithm 2 can be further improved by adjusting the checkpointing costs of files that are systematically saved (the outputs of exit tasks that are required by other superchains). When these files are large compared to the others, the dynamic programming approach may lead to fewer checkpoints than with small such files because each aggregated checkpoint cost (C_i^j) is large. However, these files are always checkpointed (to avoid crossover dependencies) and should have no impact on where to place additional checkpoints. A solution is to integrate the cost of each such checkpoint (c_{kl}) into the cost of its producing task (w_k), assuming that this checkpoint is done directly after the execution of the task, and to discard these costs from the aggregated checkpoint costs (C_i^j). This optimization is particularly useful when these necessary checkpoints are costly.

Remark #4 – We said that a superchain is checkpointed when the output data of all its exit tasks are checkpointed. However, this does not mean that these output data need to be checkpointed after the execution of the last task of the superchain. Consider the superchain in the example of Figure 2.5 with

two exit tasks T_{11} and T_{12}. Algorithm 2 systematically takes a checkpoint after the last task T_{12} but not necessarily after T_{11}. If a checkpoint is taken after T_{11}, then its output data is saved before T_{12} executes. Otherwise, this output data is saved when T_{12} completes. Both options are possible. Regardless, the structure of M-SPGs ensures that T_{11} and T_{12} have the same successors outside the superchain, and thus recovery is straightforward.

2.5 The CkptNone strategy

In this section we establish the complexity of computing the expected makespan of a scheduled task graph when the CkptNone strategy is used. In Section 2.5.1 we construct a simple instance and show that it is already #P-complete, thereby establishing the #P-completeness of the problem. Then in Section 2.5.2 we derive a simple formula to approximate the expected makespan.

2.5.1 #P-completeness

Let us define the following problem:

Definition 2.1 (DAG-MKS)**.** *Consider a task graph with n tasks. Each task T_i is scheduled on its own processor P_i and has a unitary cost. Each task can thus start executing as soon as all its predecessors have completed (there are no resource constraints). There is a fixed probability p_i that each processor P_i fails when it executes its allocated task T_i for $1 \leq i \leq n$. Once P_i has failed, it restarts at the next time-step and it cannot fail again. Hence, if P_i fails while executing T_i, it will successfully re-execute T_i during the next time-step. The problem is to compute the expected makespan of the schedule.*

In this simplified problem, we have discrete times-steps, and failures hit processors only once, similarly to the approximated execution model given in Equation 2.1. Note that with this simple model, the schedule is always executed in bounded time.

Theorem 2.1. DAG-MKS *is #P-complete.*

Proof. We show this result with a reduction from REL [46, 81], a #P-complete problem. Consider a DAG with a source vertex, and let V_i be the set of vertices with a path of length $i-1$ from the source. In the following, we consider layered graphs, and V_i is thus the set of vertices on layer i. A transportation DAG is a graph in which edges go only from the source $v_1 \in V_1$ to vertices in V_2, from vertices in V_2 to vertices in V_3 and from vertices in V_3 to the sink $v_n \in V_4$. In other words, this is a four-layer graph shaped as a directed bipartite graph with a source and a sink (see Figure 2.7).

Definition 2.2 (REL)**.** *We consider a transportation DAG with possibly multiple edges and where each edge may fail with probability p. The objective is to determine the probability that there is a path between the source and the sink.*

We first transform an instance of REL into an instance of a related problem in which the vertices fail instead of the edges. Each initial vertex remains unchanged and cannot fail. We replace each edge by a vertex that can fail with probability p and connect this vertex to the predecessor and the successor of the edge. This leads to a transformed graph with 7 layers of vertices. Vertices in even layers fail with probability p, whereas vertices in odd layers do not fail. The probability that there is a path between the source and the sink is the same as with the initial REL instance.

We now build an instance of DAG-MKS with the same graph structure, and we let $p_i = 1 - p$ for all vertices of even layers and $p_i = 0$ otherwise. We will prove that determining the probability that the makespan of this DAG is equal to 10 solves the REL instance. We introduce some notations for the REL

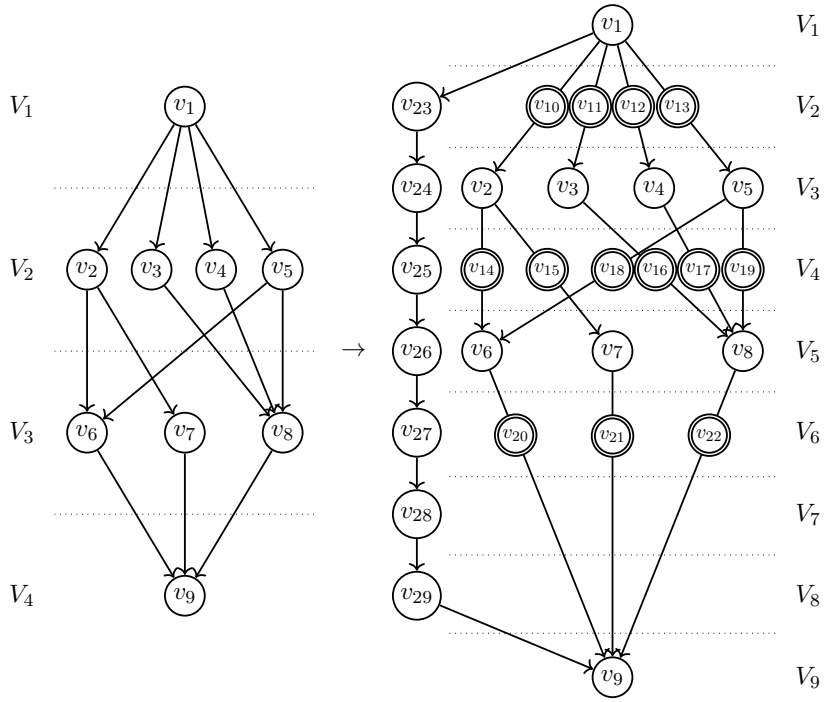


Figure 2.7: The transportation graph of the REL instance (left) and the corresponding DAG-MKS instance (right). In the REL instance, each edge may fail with probability p. In the DAG-MKS instance, tasks with a double circle (v_{10} to v_{22}) may fail with probability $1-p$, while other tasks never fail.

instance. Let E_{ij} be the event that occurs when the edge from vertex v_i to v_j succeeds ($\Pr[E_{ij}] = 1-p$). All E_{ij} are independent. Let F_i^j be the event that occurs when there is a path from the source to a vertex $v_i \in V_j$ in the REL instance. Then, F_1 always occurs, $F_i = E_{1i}$ for $v_i \in V_2$, $F_i = \bigcup_{j\in\text{Pred}(v_i)} F_j \cap E_{ji}$ for $v_i \in V_3$ and $F_n = \bigcup_{j\in\text{Pred}(v_n)} F_j \cap E_{jn} = \bigcup_{j\in\text{Pred}(v_n)} \bigcup_{k\in\text{Pred}(v_j)} E_{1k} \cap E_{kj} \cap E_{jn}$ (recall that $\text{Pred}(v_i)$ is the set of predecessors o v_i). Solving REL consists in determining $\Pr[F_n]$.

We now focus on the DAG-MKS instance. Let G_i be the event that occurs when vertex v_i in layer V_j fails at step j and is re-executed, for $j \in \{2,4,6\}$ (recall that vertices in odd layers never fail). We have $\Pr[G_i] = 1-p$, which is equivalent to the event $E_{\text{pred}(v_i)\text{succ}(v_i)}$ (we use pred, resp. succ, in lowercase to denote a single predecessor, resp. successor). All G_i are independent. Let C_i be the completion time of vertex v_i. Consider the first three layers. The event $\{C_1 = 1\}$ always occurs, because the source vertex never fails. For $v_i \in V_2$, either no fault occurs ($\overline{G_i}$) and $C_i = 2$, or a fault occurs and it takes one more time-step to execute task v_i, i.e., we derive that $G_i = \{C_i = 3\}$. Finally, $\{C_i = 4\} = \{C_{\text{pred}(v_i)} = 3\} = G_{\text{pred}(v_i)}$ for $v_i \in V_3$. Analogously, for the two next layers, we have: $\{C_i = 6\} = \{C_{\text{pred}(v_i)} = 4\} \cap G_i$ for $v_i \in V_4$ and $\{C_i = 7\} = \bigcup_{j\in\text{Pred}(v_i)}\{C_j = 6\} = \bigcup_{j\in\text{Pred}(v_i)}\{C_{\text{pred}(v_j)} = 4\} \cap G_j$ for $v_i \in V_5$. For the last two layers, we have: $\{C_i = 9\} = \{C_{\text{pred}(v_i)} = 7\} \cap G_i$ for $v_i \in V_6$ and $\{C_n = 10\} = \bigcup_{j\in\text{Pred}(v_i)}\{C_j = 9\} = \bigcup_{j\in\text{Pred}(v_n)}\{C_{\text{pred}(v_j)} = 7\} \cap G_j$. After simplification, we have $\{C_n = 10\} = \bigcup_{j\in\text{Pred}(v_n)} \bigcup_{k\in\text{Pred}(\text{pred}(v_j))} G_{\text{pred}(\text{pred}(v_k))} \cap G_k \cap G_j$. We see that $\Pr[\{C_n = 10\}] = \Pr[F_n]$ because the graph structure of the DAG-MKS instance is the same as REL.

It remains to prove that determining the probability that the makespan is 10 (i.e., $\Pr[\{C_n = 10\}]$) can be done by determining the expected makespan. We use a technique similar to the one used in [46]. We simply add a series of 7 never-failing vertices between the source and the sink, in parallel of the previous graph (see Figure 2.7). Then, the expected makespan of this new DAG is $\Pr[\{C_n = 10\}] + 9$. □

The general problem (i.e., when task costs are not unitary, when several tasks may be allocated to a given processor, when there is a probability of failure during re-execution, when there are recovery costs, etc.) is thus also #P-complete, and likely more challenging than DAG-MKS.

2.5.2 Approximating the makespan

Section 2.5.1 shows the difficulty of computing the makespan of a schedule where no task is checkpointed. Still, we can derive the following approximation:

Theorem 2.2. *Consider a schedule for an* M-SPG G *with* p *processors, with all tasks assigned to processors and no checkpoint. Let* W_{par} *be the parallel time of the schedule with no failure, and let* λ *be the processor's exponential failure rate. An approximation of the expected makespan* $EM(G)$ *is*

$$EM(G) = \left(\frac{1}{p\lambda} + d\right)(e^{p\lambda W_{par}} - 1)$$

Proof. The idea is to consider a single task of weight W_{par} and to compute its expected execution time as in Equation 2.2. The only differences are that: (i) we use the platform's Exponential failure rate $p\lambda$ [56]; and (ii) we neglect the recovery cost. □

While this formula is likely to be inaccurate, we are not aware of any better approximation. In Section 2.6, we do not use $EM(G)$ to evaluate the expected makespan of the CKPTNONE strategy; instead, we use Monte-Carlo simulations. We consider a single task of weight W_{par} and compute its expected execution time by sampling its exponential distribution with failure rate $p\lambda$. After repeating this operation for a large number of trials, each of which produced a sample makespan, we approximate the expected makespan as the average over these samples, thereby obtaining an accurate evaluation.

2.6 Experiments

In this section, we present experimental results that quantify the effectiveness of the proposed PROPCKPT algorithm.

2.6.1 Experimental methodology

Our experiments are for representative workflow applications generated by the Pegasus Workflow Generator (PWG) [13, 78, 92]. PWG uses the information gathered from actual executions of scientific workflows as well as domain-specific knowledge of these workflows to generate representative and realistic synthetic workflows (the parameters of which, e.g., total number of tasks, can be chosen). We consider three different classes of workflows generated by PWG, namely MONTAGE, LIGO and GENOME, which are all M-SPGs[2] (information on the corresponding scientific applications is available in [65, 78]):

- MONTAGE: The NASA/IPAC Montage application stitches together multiple input images to create custom mosaics of the sky. The average weight of a MONTAGE task is 10s. Structurally, MONTAGE is a three-level graph [31]. The first level (reprojection of input image) consists of a bipartite directed graph. The second level (background rectification) is a bottleneck that consists

[2] MONTAGE is not fully an M-SPG because of some transitive edges that go from the source tasks to the exit tasks of the second layer. However, this does not impact PROPCKPT because the source tasks are also exit tasks and are thus always checkpointed.

in a join followed by a fork. Then, the third level (co-addition to form the final mosaic) is simply a join.

- LIGO: LIGO's Inspiral Analysis workflow is used to generate and analyze gravitational waveforms from data collected during the coalescing of compact binary systems. The average weight of a LIGO task is 220s. Structurally, LIGO can be seen as a succession of Fork-Joins meta-tasks, that each contains either fork-join graphs or bipartite graphs (see the LIGO IHOPE workflow in [78]). Depending on the number of tasks required, PWG may not output an M-SPG Ligo workflow because of some incomplete bipartite graphs. In these cases, to ensure full fairness when comparing approaches, the baseline strategies process the original workflow while PROPCKPT processes a workflow where bipartite graphs have been extended with dummy dependencies for zero-size files (which adds synchronizations but no data transfers).
- GENOME: The epigenomics workflow created by the USC Epigenome Center and the Pegasus team automates various operations in genome sequence processing. The average weight of a GENOME task depends on the total number of tasks and is greater than 1000s. Structurally, GENOME starts with many parallel fork-join graphs, whose exit tasks are then both joined into a new exit task, which is the root of fork graphs (see the Epigenomics workflow in [78]).

We generate MONTAGE, LIGO, and GENOME workflows with various number of tasks. For each task T_i in the workflow, its weight w_i is generated by PWG. We compute the time required to read or save the data produced by task T_i and needed by task T_j, c_{ij}, by dividing the size of the file in bytes by the stable storage bandwidth in byte/sec. The file sizes are generated by PWG. In some instances, a task may generate the same file for more than one successor task. In this case a checkpoint saves the file only once.

In the experiments we consider different exponential processor failure rates. To allow for consistent comparisons of results across different M-SPGs (with different numbers of tasks and different task weights), we simply fix the probability that a task fails, which we denote as p_{fail}, and then simulate the corresponding failure rate. Formally, for a given M-SPG, $G = (V, E)$ and a given p_{fail} value, we compute the average task weight as $\bar{w} = \sum_{i \in V} w_i / |V|$, where w_i is the weight of the i-th task in V. We then pick the failure rate λ such that $p_{\text{fail}} = 1 - e^{-\lambda \bar{w}}$. We conduct experiments for three p_{fail} values: 0.01, 0.001, and 0.0001.

An important factor that influences the performance of checkpointing strategies, and more precisely of the checkpointing and recovery overheads, is the data-intensiveness of the application. The workflows generated by PWG give task durations in seconds and file sizes in bytes, which makes it difficult to quantify data-intensiveness. Instead, we define the Communication-to-Computation Ratio (CCR) as the time needed to store all the files handled by a workflow (input, output, and intermediate files) divided by the time needed to perform all the computations of that workflow on a single processor. The total store time is the total file size divided by the bandwidth to the stable storage. Instead of picking arbitrary bandwidth values, which would have different meanings for different workflows, we vary the CCR by scaling file data sizes by a factor. This makes it possible to consider and quantify the data-intensiveness of all workflows in a coherent manner across experiments and workflow classes and configurations.

The experiments compare PROPCKPT to the two extreme approaches, CKPTALL and CKPTNONE. For all strategies, we use Monte-Carlo simulations [73, 93] to compute the expected makespan of the solutions. A task in the DAG succeeds or fails as determined by sampling the exponential time-to-failure distribution, and a task can fail more than once. After sampling, the DAG is deterministic and its makespan can be computed as the length of its longest path. This operation is repeated for a large number of trials, each of which produces a sample makespan. These samples approach the actual makespan distribution as the number of trials increases. Following [25], we use 300,000 trials and approximate the expected makespan as the average over the resulting 300,000 makespan samples. This enormous

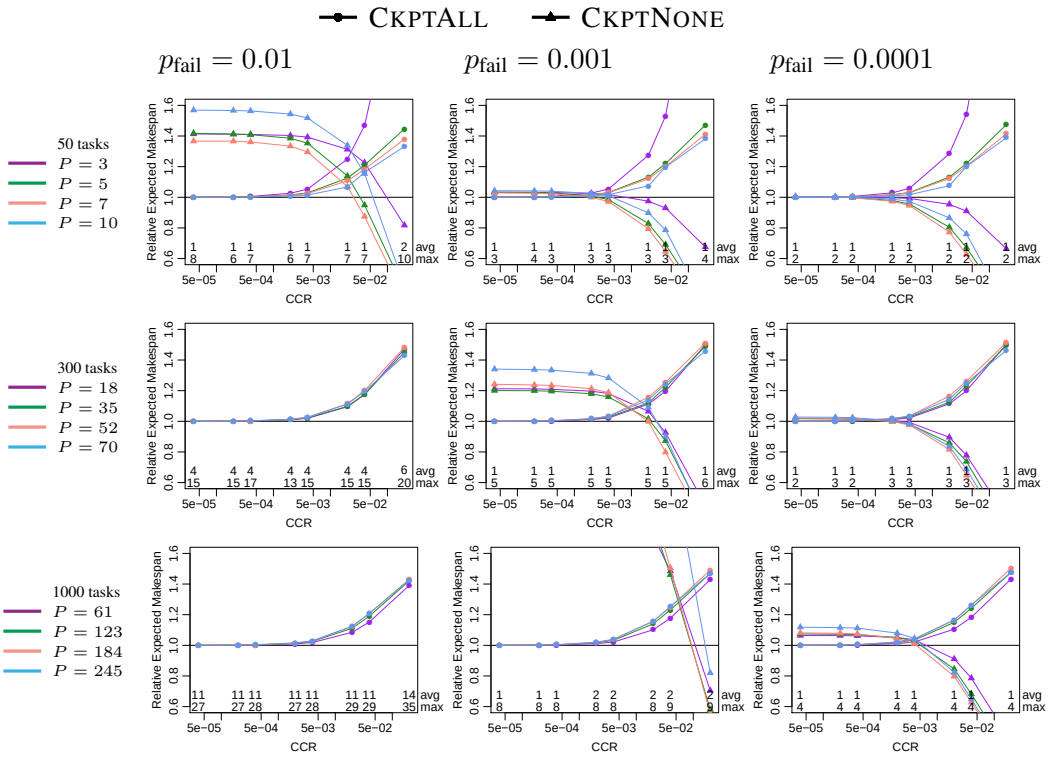


Figure 2.8: Expected makespan of CkptAll and CkptNone relative to that of PropCkpt for the Genome workflow, three different failure rates, three workflow sizes, and varying Communication-to-Computation Ratio (CCR).

number of trials is prohibitively expensive in practice, but provides us with an accurate ground truth to compare the different strategies. Code is publicly available at [47].

2.6.2 Expected makespan

In this section, we compare the expected makespan of two baseline strategies (CkptAll and CkptNone) over that of our proposed strategy (PropCkpt). Figures 2.8 to 2.10 show expected makespans for CkptAll and CkptNone relative to that of PropCkpt vs. Communication-to-Computation Ratio (CCR). Data points above the $y = 1$ line denote cases in which our strategy outperforms a competitor (i.e., achieves a lower expected makespan). Each figure shows results for workflows with 50, 300 and 1000 tasks, for various numbers of processors P, and for the three p_{fail} values (0.01, 0.001, and 0.0001). We report the average and maximum number of failures that occur for the 300,000 trials of each execution. These numbers are shown above the horizontal axis in each figure, labeled as *avg* and *max*. For different workflows, we pick the number of processors, P, as follows: we compute the maximum parallelism of the workflow, p, and pick $P \in \{p/4, p/2, 3p/4, p\}$.

A clear observation is that PropCkpt always outperforms CkptAll.[3] In each scenario, above some CCR value, which depends on the failure rate and the workflow size, PropCkpt leads to significant improvement over CkptAll. As the CCR decreases, the relative expected makespan of CkptAll

[3]There are in fact a couple of CCR values for Ligo with 300 tasks for which this is not true. This is an artifact of our slight transformation of the Ligo workflow (see Section 2.6.1 for details).

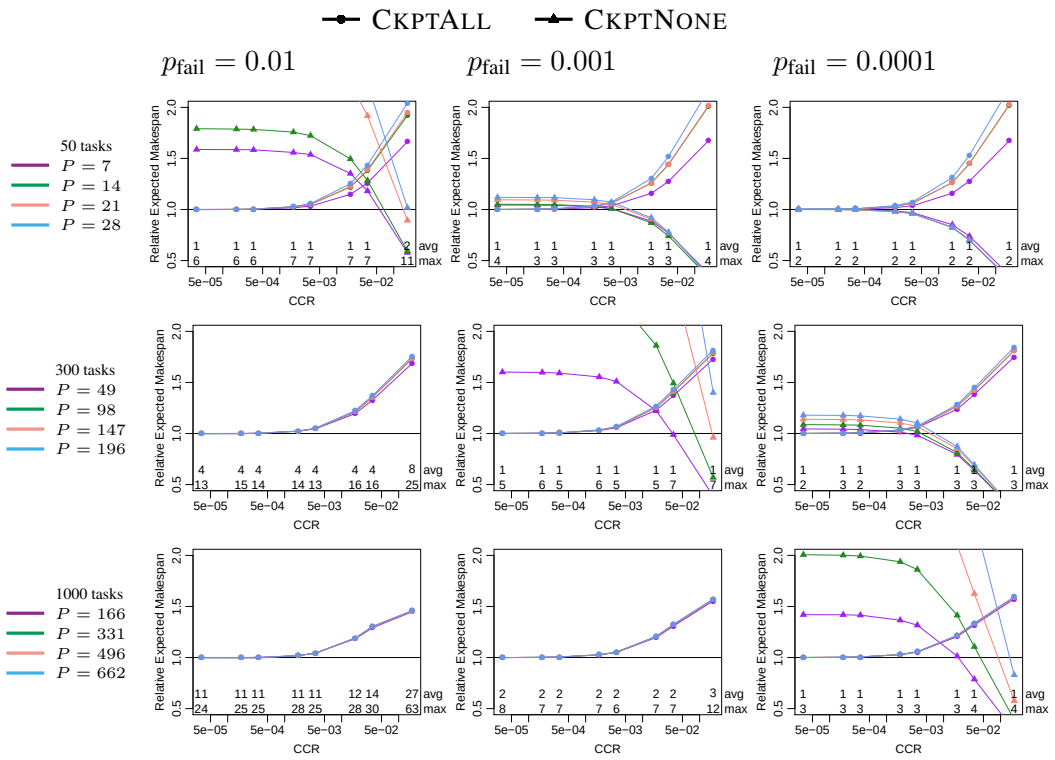


Figure 2.9: Expected makespan of CkptAll and CkptNone relative to that of PropCkpt for the Montage workflow, three different failure rates, three workflow sizes, and varying Communication-to-Computation Ratio (CCR).

decreases and converges to 1. This is because when checkpointing becomes cheap enough PropCkpt decides to checkpoint every task, and thus is equivalent to CkptAll.

Another common trend is that the relative expected makespan of CkptNone increases as the CCR decreases since as checkpoints become cheaper not checkpointing becomes a losing strategy (poorer resilience to failures, but little saving on checkpointing overhead). Overall, CkptNone becomes worse whenever there are more failing tasks, i.e., when the failure rate increases (going from the rightmost column to the leftmost one in the figures), and/or when the number of tasks increases (going from the topmost row to the bottom one in the figures). When the failure rate is high and the workflows are large (the bottom left corner of the figures), the relative expected makespan of CkptNone is so high that it does not appear in the plots.

PropCkpt achieves better results than CkptNone except when (i) checkpoints are expensive (high CCR) and/or (ii) failures are rare (low p_{fail}). In these cases, checkpointing is a losing proposition, and yet PropCkpt by design always checkpoints some tasks (it checkpoints all exit tasks of superchains). In practice, in such cases, the optimal approach is to bet that no failure will happen and to restart the whole workflow execution from scratch upon the very rare occurrence of a failure. The results above for our particular benchmark workflows, and our experimental methodology in general, make it possible to identify these cases so as to select which approach to use in practical situations.

Besides the results shown in Figures 2.8 to 2.10, we performed more comprehensive experiments to assess the performance of our strategy PropCkpt on a larger workflow set, e.g., Genome and Montage with 100, 500 and 700 tasks, Ligo with 100 and 400 tasks [50]. All results show the same trends as the results presented in this section.

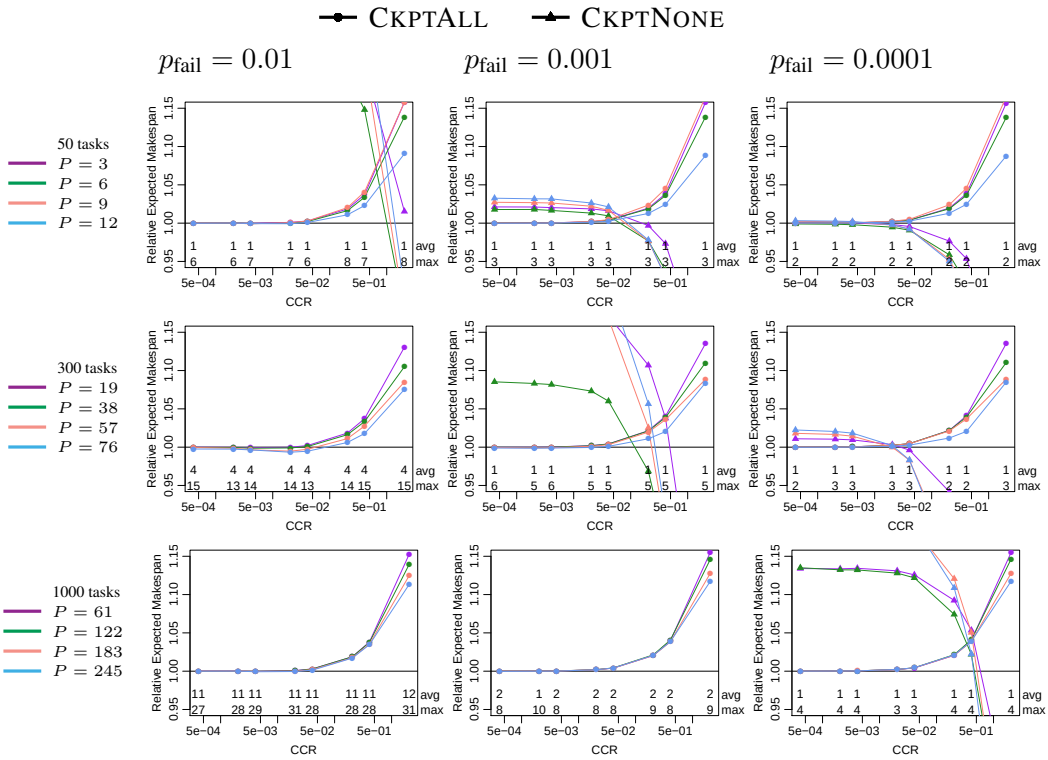


Figure 2.10: Expected makespan of CkptAll and CkptNone relative to that of PropCkpt for the Ligo workflow, three different failure rates, three workflow sizes, and varying Communication-to-Computation Ratio (CCR).

2.7 Conclusion

We have proposed a scheduling/checkpointing algorithm, called PropCkpt, for executing workflow applications (M-SPGs) on parallel computing platforms in which processors are subject to fail-stop failures. The objective function to be minimized is the expectation of the makespan, which is a random variable due to probabilistic task re-executions due to failures. For general Directed Acyclic Graphs (DAGs), this problem is intractable and even computing the objective function is itself a #P-complete problem. However, by restricting our work to a class of structured recursive DAGs, Minimal Series-Parallel Graphs (M-SPGs), which are broadly relevant to production workflow applications, we are able to design a sensible algorithm and to accurately compute the expected makespan of the solutions it produces. A competing approach, CkptAll, side-steps part of the difficulty of solving the problem by saving all application data to stable storage so as to minimize the impact of failures, with the drawback of maximizing checkpointing overhead. This is the approach employed by default in most production workflow executions, in which each task is an executable that reads all its input from files and writes all its output to files. Another competing approach, CkptNone, is a risky zero-overhead approach in which the entire workflow is re-executed from scratch in case of a failure. The broad objective of our algorithm is to produce solutions that strike a good compromise between these two extremes. For the CkptNone approach, when applied to general DAGs, we have established that the problem of computing the expected makespan is #P-complete, which to the best of our knowledge is a new result.

We have evaluated the effectiveness of our algorithm by considering realistic workflow configurations produced by a workflow generator from the Pegasus community [13, 78, 92]. We have shown

that our PROPCKPT algorithm does indeed provide an attractive compromise between the CKPTALL and CKPTNONE approaches. More specifically, PROPCKPT always outperforms CKPTALL and is only outperformed by CKPTNONE when checkpoints are expensive and/or failures are rare. Our experimental methodology provides the quantitative means to identify these cases (based on application CCR, platform scale, and failure rates), so as to select which approach to use in practice.

Next step in Chapter 3, we will extend these techniques to enable the approach for arbitrary workflows. We point out that PROPCKPT can be straightforwardly extended to deal with General Series Parallel Graphs, which are defined in [104] as graphs whose transitive reductions are M-SPGs.

Another promising direction is to refine the linearization algorithm for superchains (Algorithm 1). Instead of choosing the topological sort arbitrarily, one may try and reduce the total volume of output files, in the hope of reducing the total checkpointing cost when applying Algorithm 2 after the linearization. This problem is related to the sum cut problem [34], which is NP-complete for general DAGs, but may be amenable to efficient solutions for M-SPGs.

Finally, it may be possible to improve the overall approach by making allocation and checkpointing decisions simultaneously, possibly extending branch and bound methods for scheduling DAGs. This is a challenging proposition, and for now a solution seems out of reach.

Chapter 3

Generic approaches for arbitrary task graphs

In the previous Chapter 2, we took advantage of the recursive structure of M-SPGs and used proportional mapping [80] for scheduling and checkpointing M-SPG workflows as sets of superchains, which leads to an optimal checkpointing strategy (given the mapping and all exit tasks of superchains are automatically checkpointed). In this following chapter, we propose a generic approach for arbitrary task graphs. Given any DAG and a set of processors on which fail-stop failures strike with Exponentially distributed inter-arrival times, the objective is to schedule the task executions and potential checkpoints such that the expected completion time (or makespan) is minimized. For general graphs, we have to resort to classical scheduling heuristics such as HEFT [101] and MinMin [18], two reference scheduling algorithms widely used by the community. We provide extensions of HEFT and MinMin that allow for a smaller subset of tasks to be checkpointed and lead to better makespans than the versions where each task (CkptAll) or no task (CkptNone) is checkpointed.

The main contributions of this chapter are the following:

- We deal with arbitrary dependency graphs, and require no graph transformation before applying our scheduling and checkpointing algorithms;
- We compare several mapping strategies and combine them with several checkpointing strategies;
- We design an event-based simulator to evaluate the makespan of the proposed solution (described in Section 3.3.2). Indeed, computing the expected makespan of a solution is a difficult problem (as shown in Section 2.2.5), and simple Monte-Carlo based simulations cannot be applied to general DAGs unless all tasks are checkpointed: otherwise, sampling the weight distribution for each task independently is not enough to compute the makespan, since a failure may involve re-executing several tasks (as shown in Section 3.1);
- We report extensive experimental evaluation with both real-world and randomly generated workflows to quantify the performance gain achieved by the proposed approach.

The rest of the chapter is organized as follows. First in Section 3.1, we work out an example to understand the difficulties in general graphs. Then we detail our scheduling and checkpointing algorithms in Section 3.2. We give experimental results in Section 3.3. Finally, we provide concluding remarks and directions for future work in Section 3.4. Please refer to Section 2.2.1 and Section 2.2.2 for the description of the execution model and the fault-tolerance model respectively.

3.1 Example

In this section, we illustrate the difficulty of deciding where to place checkpoints in a general workflow. Consider the example of Figure 3.1 with 9 tasks, T_i, $1 \leq i \leq 9$, that have been mapped on 2 processors as shown on the figure. Note that this DAG cannot be reduced to an M-SPG and the previous approach

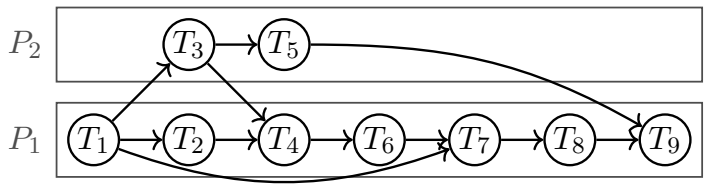


Figure 3.1: Schedule of a workflow with 9 tasks on 2 processors (each edge corresponds to a file dependence between tasks).

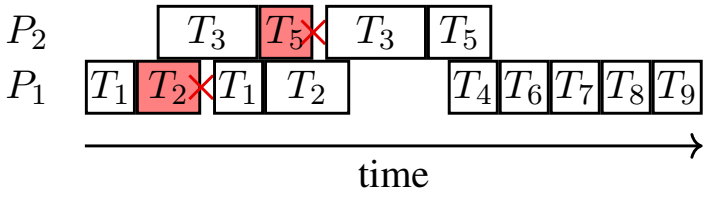


Figure 3.2: Sample execution of the workflow in Figure 3.1 without any checkpoint, with two failures striking during the execution of T_2 on P_1 and during that of T_5 on P_2.

in Chapter 2 cannot be applied for this graph. While most tasks are assigned to processor P_1, some tasks are assigned to the second processor, P_2, to exploit the parallelism of the DAG. Any dependency between two tasks represents a file that is required to start the execution of the successor task; hence, $T_1 \rightarrow T_2$ represents a file produced by task T_1 that is required for the execution of task T_2 to start. Because T_1 and T_2 are both executed on processor P_1, this file is kept in the memory of P_1 after T_1 completes. However, for the dependency $T_1 \rightarrow T_3$, because the tasks T_1 and T_3 are executed on different processors, the corresponding file must be retrieved by P_2. Such a dependency between two tasks assigned to two different processors is called a *crossover dependency* as stated in Section 2.1.

In a first scenario, let us suppose that no task is checkpointed as showed in Figure 3.1: then if no failure strikes, the makespan will be the shortest possible, consisting only of the execution time of each task and of retrieving the necessary input files. However, as soon as a failure happens, we may need to restart the whole application from the very beginning. To study such a scenario, we need to explicit the memory management. Let us assume that once a processor has sent a file to another processor, then this file is deleted from the memory of the producing processor. For instance, as soon as P_2 has received from P_1 the file corresponding to the dependency $T_1 \rightarrow T_3$, this file is erased from the memory of P_1. Remember that a failure wipes out the whole content of the memory of the struck processor. Thus, if a failure strikes during the execution of T_5, to be able to re-attempt to execute T_5, T_3 will need to be re-executed before (because the file $T_3 \rightarrow T_5$ is no longer available), which requires T_1 to be re-executed first (because the file $T_1 \rightarrow T_3$ is no longer available). Hence, a single failure in a part of the graph may

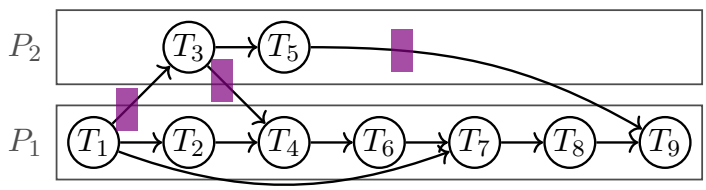


Figure 3.3: A purple *crossover* checkpoint is performed for each file produced by one processor and used by another one.

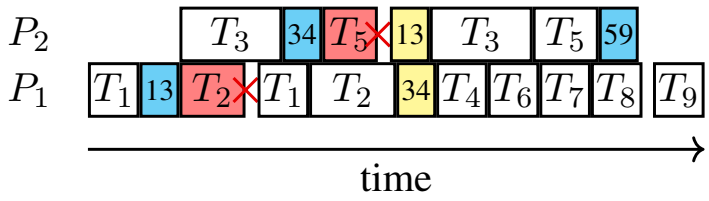


Figure 3.4: Sample execution of the application in Figure 3.3 with two failures striking during the execution of T_2 on P_1 and that of T_5 on P_2, with crossover checkpoints. Label ij indicates the file from T_i to T_j. Now T_4 can start before the re-execution of T_3 since its output was checkpointed.

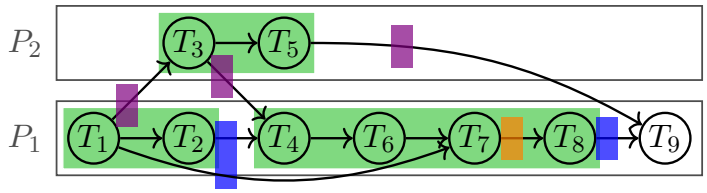


Figure 3.5: Blue *induced* checkpoints are used to isolate task sequences on a processor (labeled in green, such as the sequence T_4, T_6, T_7 and T_8 on P_1). Finally, additional checkpoints can be added inside an idle-free task sequence through a dynamic programming algorithm: the orange checkpoint corresponds to such an addition.

require the re-execution of most of the workflow, which is known as the domino effect in fault tolerance protocols and needs to be mitigated. Figure 3.2 shows an example of execution of the DAG when no task is checkpointed. To execute T_4, we need both T_2 and T_3 to finish successfully, and that no fault strikes neither P_1 nor P_2 between the completion of these tasks and the start of T_4. Here, T_2 does not finish so T_1 is re-executed. When P_2 fails, we need to re-execute T_3, which requires input from T_1. Luckily (!), P_1 already suffered from a failure, so T_1 has already been re-executed. Otherwise, we would have had to restart the execution of the whole workflow because of the failure of P_2.

To avoid rolling back to the beginning in case of failures, we can try to place some checkpoints inside the workflow. As commonly assumed in workflow management systems [1, 2, 32, 38, 109, 110], any file produced by one processor and required by another processor is necessarily saved to stable storage. Thus, an error on one processor will not lead to re-execution on another processor. In the second scenario shown in Figure 3.3, we decide to checkpoint every crossover dependency (files from T_1 to T_3, T_3 to T_4, and T_5 to T_9). An execution of that schedule is shown in Figure 3.4. Cyan boxes represent file checkpoints while yellow boxes represent data being read. We can see that thanks to the crossover checkpoints, T_4 does not need to wait for the completion of the second execution of T_3 anymore, as the output data from T_3 to T_4 has already been checkpointed. Moreover, if only a failure on P_2 happened, instead of rolling back to task T_1 to re-execute T_3 as it was the case before, T_3 could have restarted directly (although the entire content of the processor memory is lost, so all inputs of T_3 must be recovered from stable storage after a downtime before the execution of T_3 can restart). Note that we start executing the target task of a crossover dependency after checkpointing, to make sure that all files needed are available in the stable storage in case of failure. The motivation to checkpoint all files involved in crossover dependencies is to isolate the processors. Indeed, if all crossover files are checkpointed, a failure on a processor will never lead to the re-execution of a task successfully executed on another processor. Overall, we will lose less time recomputing tasks or waiting for their second completion. However, reading from stable storage and checkpointing also take time. Same as the previous chapter,

finding the right trade-off is the main focus: deciding which tasks should be checkpointed, so that the overhead added by the checkpointing and reading of files is not more expensive than the re-execution of tasks.

We conclude by informally introducing examples of checkpointing strategies that achieve desirable trade-offs (see Section 3.2.2 for details). Two additional checkpoints, in blue, called *induced* task checkpoints, have been added in Figure 3.5. Their role is to secure the fast re-execution of tasks that are the target of a crossover dependency, namely T_4 and T_9. The blue task checkpoint after T_2 isolates the execution of the task sequence $S_1 = \{T_4, T_6, T_7, T_8\}$ on P_1. To this purpose, it is necessary to checkpoint all intermediate results that may be used after the execution of T_2 (also refer to the definition of a task checkpoint in Section 2.4.1): these are the files generated by previous tasks, namely $T_1 \rightarrow T_7$ and $T_2 \rightarrow T_4$. This way, when a failure strikes, previous tasks do not have to be restarted and the computation may be restarted directly from T_4. This way, tasks in the sequence S_1 may be sequentially executed without idle time. It would not have been possible to include T_1 and T_2 in S_1 because T_4 could have waited for the completion of T_3 leading to idle time in some scenarios. Similarly, the second blue task checkpoint isolates the execution of T_9.

Finally, once the four tasks T_4, T_6, T_7 and T_8 of the sequence S_1 have been "isolated" from other tasks, it is possible to use a dynamic programming algorithm similar to that proposed in Section 2.4.2 in order to introduce additional task checkpoints. In the example of Figure 3.5, a single additional task checkpoint, in orange, is inserted after T_7.

3.2 Scheduling and checkpointing algorithms

In this section, we first present heuristics to map tasks to processors. Then we propose three different checkpointing strategies that can be used simultaneously.

3.2.1 Scheduling heuristics

We map tasks to processors and schedule them using two classical scheduling heuristics, HEFT [101] and MINMIN [18]. We run these heuristics as if the platforms were not subject to failures, that is, without considering checkpoints. Therefore, we decide first on which processor a task will be executed, and the order in which a processor will execute tasks, before deciding when and what to checkpoint (see Section 3.2.2). However, we present variants of HEFT and MINMIN, named HEFTC and MINMINC, that are specifically designed for our failure-prone framework.

Heterogeneous Earliest Finish Time first (HEFT) is presented as the HEFTC variant in Algorithm Algorithm 3. The original HEFT algorithm comprises two phases. In a first *task prioritizing phase*, the bottom-level of all tasks is computed and tasks are ordered by non-increasing bottom-levels. The bottom-level of a task is the maximum length of any path starting at the task and ending in an exit task, considering that all communications take place [29]. In the second *processor selection phase*, the first unscheduled task is scheduled as early as possible on a processor that minimizes its completion time. In all cases, ties are broken arbitrarily. To these original two phases, we add a third one, the *chain mapping phase* (Line 7 and Line 8 of Algorithm 3). If the newly mapped task T is the head of a chain in the task graph, which means T only has one successor whose only predecessor is T, then this whole chain is mapped on the same processor as T, and the tasks will be executed consecutively. Ensuring that entire chain of tasks are scheduled on the same processor decreases the number of crossover dependencies and thus, the time to checkpoint them. HEFTC has a complexity of $O(n^2)$ for a workflow with n tasks. During the processor selection phase, the earliest finish time of a task is computed in HEFTC

while assuming that the newly mapped task must start *after* all tasks previously scheduled on that processor have completed. On the contrary, the original HEFT heuristic is allowed to perform backfilling following a classical insertion-based policy, as long as the completion time of no task is delayed. Allowing backfilling is more expensive at scheduling time but should lower the execution time (the complexity of HEFT with backfilling is also $O(n^2)$ with homogeneous processors). We do not allow backfilling for HEFTC because it could be antagonistic to the chain mapping phase if it led to backfill the head of the chain, but not the whole chain.

Algorithm 3: HEFTC

```
1 Compute the bottom-level of all tasks by traversing the graph from the exit tasks
2 Sort the tasks by non-increasing values of their bottom-levels
3 while there are unscheduled tasks do
4 |   Select the first task T_i
5 |   k ← arg min_{1≤k≤p} EarliestFinishTime(T_i, P_k)
6 |   Schedule task T_i on processor P_k
7 |   if T_i is the head of a chain of tasks then
8 |   |   Schedule the whole chain continuously on P_k
```

The MINMIN scheduling algorithm is presented in the MINMINC variant in Algorithm 4. The original MINMIN algorithm is a simple loop which, at each step, schedules the task that can finish the earliest among unscheduled tasks. Therefore, at each step it considers all ready tasks and, for each of them, all the processors. We (try to) improve this heuristic by adding a *chain mapping phase* exactly as previously (Line 5 and Line 6 of Algorithm 4). MINMINC has a complexity of $O(n^2p)$ for a workflow with n tasks and p processors.

Algorithm 4: MINMINC

```
1 ReadyTasks ← entry tasks
2 while there are unscheduled tasks do
3 |   Pick a task T ∈ ReadyTasks and a processor P such that the completion time of T on P is
  |     minimum among the Earliest Finish Times of all ready tasks
4 |   Schedule task T on processor P
5 |   if T is the head of a chain of tasks then
6 |   |   Schedule the whole chain continuously on P
7 |   Update ReadyTasks
```

3.2.2 Checkpointing strategies

While the previous scheduling algorithms provide mappings of tasks to processors, it remains to decide which files must be checkpointed and when. This section introduces finer strategies than the two extremes solutions that consist of checkpointing no task or all tasks. These two extreme solutions, CKPTNONE and CKPTALL, are denoted with the suffixes NONE and ALL, respectively.

The minimum strategy that is required to isolate processors consists in checkpointing all files (*file checkpoints*) that must be transferred between any pair of processors, i.e., exactly the files corresponding to crossover dependencies. In this case, any failure on a processor will not require any re-execution on other processors. The strategy is denoted with a "C" in the checkpoint suffix.

For the next two additional strategies, we apply *task checkpoints* as proposed in Section 2.4. While a simple file checkpoint consists of writing to stable storage a file that corresponds to a dependency between two tasks, a task checkpoint consists of writing all files that (i) reside in memory on a processor; (ii) will be used later by tasks assigned to the same processor; and (iii) have not already been checkpointed. In the example in Section 3.1, for each crossover dependency we did a simple file checkpoint rather than a full task checkpoint. A task checkpoint after task T_3 would have also checkpointed the file corresponding to the dependency $T_3 \rightarrow T_5$. A non-trivial task checkpoint for the example of Section 3.1 would be a task checkpoint for task T_2. This checkpoint would require checkpointing the files corresponding to the dependencies $T_2 \rightarrow T_4$ and $T_1 \rightarrow T_7$.

When a task checkpoint is performed after the execution of a task, multiple files may be checkpointed "at the same time" (either newly created files or previously created ones that will later be used). If several files are checkpointed, they are all checkpointed after the task completion, one after the other (in any order). When absent from memory (following a failure), input files are read from stable storage as late as possible, just before the execution of the task that needs them.

Checkpointing crossover dependencies enables to isolate processors, in that there is no re-execution propagation from a processor to another. However, when a task is the target of a crossover dependency, its starting time is the maximum of the availability times of all its input files, and these files come from different processors. Therefore, its starting time may be delayed by failures occurring on other processors. Because failures can strike during idle time, it may be beneficial to try to use the potential waiting time by performing a task checkpoint of the task preceding the target task. This way, the whole content of the memory will be preserved, the cost of the checkpoint may be offset by some waiting time, and if a failure strikes during the remaining waiting time all input files remain available. Therefore, we propose a new checkpointing strategy denoted with "I" in the checkpoint suffix. This strategy consists of checkpointing all *induced* dependencies. A dependency $T_i \rightarrow T_j$ is an *induced* dependency if T_i and T_j are scheduled on the same processor P and there exists a crossover dependency $T_k \rightarrow T_l$ such that T_l is scheduled on P after T_i and before T_j (or $T_l = T_j$). Checkpointing these induced dependencies is done by performing a task checkpoint of the task preceding T_l on P. In the example of Section 3.1, the dependencies $T_2 \rightarrow T_4$ and $T_1 \rightarrow T_7$ are both induced dependencies because of the crossover dependency $T_3 \rightarrow T_4$. As a side note, we point out that our *induced* task checkpoints are not related to those introduced in [8] to track consistency among execution traces of general applications: we deal with simple task graphs and can easily regenerate missing data by following the dependencies.

So far, we have only introduced checkpoints to isolate processors, either to avoid failure propagation or to try to minimize the impact of processors having to wait from each other. We further consider checkpoints that more directly optimize expected total execution time. We present an additional strategy, denoted by the suffix "DP", which adds additional checkpoints through a $O(n^2)$ dynamic programming algorithm, which is a transposition of that of Section 2.4.2. This dynamic program considers a maximal sequence of consecutive tasks that are all assigned to the same processor, and that are isolated from other tasks: the sequence contains no checkpoint and none of its tasks is the target of a crossover dependency, except for its first task. Let $T_a, \ldots, T_j$ be such a sequence of tasks. By definition, all input data produced by some previous tasks have been checkpointed. $\mathcal{E}Time(j)$ is the optimal expected time to successfully execute this sequence, when a checkpoint is taken immediately after T_j completes (with possibly earlier checkpoints). Please refer to Section 2.4.2 for the detailed formulation of $\mathcal{E}Time(j)$. Here we point out that as stated in Section 2.2.2, the calculation of $T(i,j) = \left(\frac{1}{\lambda} + d\right)\left(e^{\lambda(R_i^j + W_i^j + C_i^j)} - 1\right)$ is an upper bound, where R_i^j (resp. W_i^j and C_i^j) is the sum of the recovery (resp. execution and checkpointing) costs of tasks T_i to T_j. The recovery costs concern all input files of these tasks that are on the stable storage, while the checkpointing costs concern all files that will be checkpointed when a task checkpoint is done

after T_j. Although some files may be locally available and other files in the stable storage should be read as late as possible (just before the execution of the task that needs them), this is a necessary condition to be able to reuse, in some way, the dynamic programming approach of Section 2.4.2. This algorithm requires, by construction, that induced dependencies be checkpointed. However, we heuristically use it even when this condition is not satisfied. In this case, we take a maximal sequence while allowing tasks to be the target of crossover dependencies, and behave as if these crossover dependencies were not existing: we discard any potential waiting time that may be due to these crossover dependencies (because we have no means to estimate them).

3.3 Experiments

In this section, we describe the experiments conducted to assess the efficiency of the checkpointing strategies. In Section 3.3.1, we describe the parameters and applications used during our experimental campaign, then in Section 3.3.2 we present the simulator used to run the applications and simulate the behavior of large-scale platforms. Finally, we present our results in Section 3.3.3.

3.3.1 Experimental methodology

We consider workflows from real-world applications, namely representative workflow applications generated by the Pegasus Workflow Generator (PWG) [13, 65, 92], as well as the three most classical matrix decomposition algorithms (LU, QR, and Cholesky) [26], and randomly generated DAGs from the Standard Task Graph Set (STG) [99].

Pegasus workflows. PWG uses the information gathered from actual executions of scientific workflows as well as domain-specific knowledge of these workflows to generate representative and realistic synthetic workflows (the parameters of which, e.g., the total number of tasks, can be chosen). We consider all of the five workflows [78] generated by PWG, including three M-SPGs (GENOME, LIGO, and MONTAGE introduced in Section 2.6.1) that are used to compare our new general approach with PROPCKPT, the strategy for M-SPGs proposed in Chapter 2. Other two workflows, namely CYBERSHAKE and SIPHT, are more general DAG that we used to evaluate the performance of the strategies in this chapter.

- CYBERSHAKE: The CYBERSHAKE workflow is used by the Southern Calfornia Earthquake Center to characterize earthquake hazards in a region. The average weight of a CYBERSHAKE task is 25s. Structurally, the CYBERSHAKE workflow starts with several forks. Then each of the forked tasks has two dependencies: one to a single task (join) and one to a specific task for each of the tasks. Finally, all these new tasks are joined without another dependency this time.
- SIPHT: The SIPHT workflow, from the bioinformatics project at Harvard, is used to automate the search for untranslated RNAs (sRNAs) for bacterial replicons in the NCBI database. The average weight of a SIPHT task is 190s. Structurally, the SIPHT workflow is composed of two different parts that are joined at the end: the first one is a series of join/fork/join, while the other is made of a giant join.

We generate these workflows with 50, 300, and 700 tasks (these are the number of tasks given to the generator, the actual number of tasks in the generated workflows depend on the workflow shape). The task weights and file sizes are generated by PWG. In some instances, a single file may be used by more than one task and a dependency may represent multiple files to transfer between two tasks. In the first case, whenever a file is common to multiple dependencies, the file is only saved once. In the second case, files are aggregated into a single one.

Matrix factorizations. We consider the three most classical factorizations of a $k \times k$ tiled matrix: LU, QR, and Cholesky factorizations.

- The LU decomposition is the factorization of any matrix into a product of one lower-triangular (L) and one upper-triangular (U) matrices. Structurally, the DAG is made of k steps, with at step i, one task having two sets of $k - i - 1$ children, and each pair of tasks between the two sets having another child.
- The QR decomposition is the decomposition of a matrix into a product of an orthogonal matrix (Q) and upper-triangular matrix (R), i.e., $A = QR$ with $QQ^T = Id$. Structurally, the QR decomposition looks like the LU decomposition but it has more complex dependencies between the $k - i - 1$ children at step i.
- Cholesky is a factorization of a positive and definite matrix into the product of a triangular matrix and its transpose, i.e., $A = BB^T$ where B is lower-triangular and has non-zero values of the diagonal. The Cholesky decomposition DAG is the representation of a panel algorithm and can be constructed recursively by removing the first row and the first column of submatrices, to keep factorizing the trailing matrix.

For each factorization, we perform experiments with $k = 6$, 10, and 15, for a total of $3 \times 3 = 9$ DAGs with up to 1240 tasks. The number of vertices in the DAG depends on k as follows: the Cholesky DAG has $\frac{1}{3}k^3 + O(k^2)$ tasks, while the LU and QR DAGs have $\frac{2}{3}k^3 + O(k^2)$ tasks. There are 4 types of tasks in LU, QR, and Cholesky, which are labeled by the corresponding BLAS kernels [26], and their weights are based on actual kernel execution times as reported in [5] for an execution on Nvidia Tesla M2070 GPUs with tiles of size $b = 960$.

Random graphs. The STG benchmark [99] includes 180 instances for each size of DAGs (from 50 to 5 000). This set is often used in the literature to compare the performance of scheduling strategies. Instead of choosing part of the instances for each size, we did experiments on all instances of size 300 and 750. For each instance, one of the four DAG generators specifies the structure of the dependencies (e.g., layer-by-layer) and one of the six cost generators provides the distribution of the processing times (e.g., uniform).

Failure distribution. In the experiments, we consider different exponential processor failure rates. As what we did in Section 2.6.1, to allow for consistent comparisons of results across different DAGs (with different numbers of tasks and different task weights), we simply fix the probability that a task fails, which we denote as p_{fail}, and then simulate the corresponding failure rate. Formally, for a given DAG, $G = (V, E)$ and a given p_{fail} value, we compute the average task weight as $\bar{w} = \sum_{i \in V} w_i / |V|$, where w_i is the weight of the i-th task in V. We then pick the failure rate λ such that $p_{\text{fail}} = 1 - e^{-\lambda \bar{w}}$. We conduct experiments for three p_{fail} values: 0.01, 0.001, and 0.0001.

Checkpointing costs. An important factor that influences the performance of checkpointing strategies, and more precisely of the checkpointing and recovery overheads, is the data-intensiveness of the application. We define the Communication-to-Computation Ratio (CCR) as the time needed to store all the files handled by a workflow (input, output, and intermediate files) divided by the time needed to perform all the computations of that workflow on a single processor. For Pegasus workflows, LU, QR, and Cholesky, we vary the CCR by scaling file sizes by a factor as previously. As STG only provides task weights, we compute the average communication cost as $\bar{c} = \bar{w} \times CCR$. Communication costs are generated with a lognormal distribution with parameters $\mu = \log(\bar{c}) - 2$ and $\sigma = 2$ to ensure an expected value of $\bar{c}$. This distribution with parameter $\sigma = 2$ has been advocated to model file sizes [35]. This allows considering and quantifying the data-intensiveness of all workflows in a coherent manner across experiments and workflow classes and configurations.

Reference strategies. In the experiments, we compare our strategies to the two extreme approaches CKPTALL and CKPTNONE. We use the simulator described in Section 3.3.2. For each parameter setting

of each workflow, we run 10,000 random simulations and approximate the makespan by the observed average makespan.

3.3.2 Simulator

In order to evaluate the performance of our strategies, we implemented a discrete event simulator. The C++ code for the simulator is available at `http://github.com/vlefevre/task-graph-simulation`. To simulate the execution of applications on large-scale platforms, we operate in three steps:

1. We first read an input file describing the task-graph and the scheduling/mapping strategy;
2. Then we generate a set of fail-stop error times for each processor during a time `horizon` (that is set by the user);
3. Finally, we execute ready tasks by mapping them to a processor and we keep doing this until all tasks are executed.

The first part is basically reading a file that describes the following important elements for the simulation:

- For each task,
  - its ID,
  - its weight (i.e., duration),
  - the ID of the processor it has been mapped to,
  - several booleans indicating whether the task has to be checkpointed or not, one for each checkpointing strategy.
- For each dependency between two tasks,
  - the ID of the parent,
  - the ID of the child,
  - the list of files with their time to be loaded/written that creates the dependency (i.e., there are some of the output files of the parent and some of the input files of the child).
- For each processor, its schedule: a list of tasks that have been mapped to it and that respects the causal order of the task-graph.

The second part is done by using the inversion sampling method: we generate error times according to a random variable that follows an exponential distribution, and this exponential distribution is generated from an (assumed) uniform distribution between 0 and 1 obtained by calling the C function `rand()`, and dividing its result by the C constant `RAND_MAX`. In our case, if U is a random variable following a uniform distribution between 0 and 1, then $-\frac{\log U}{\lambda}$ follows an exponential distribution of parameter λ. We generate errors on each processor, until the time of one error is greater than the `horizon` parameter. In the experiments, it was set to at least 2 times the expected makespan we have with the CKPTALL strategy, which we computed using the Monte-Carlo method. In practice, most of the simulations were done before the horizon was reached except for NONE with large p_{fail}.

For the last step, we keep a global time t on all the processors, and we generate events happening on each processor (either a failure or the successful completion of a task). Each processor holds the time of its last event in a variable t_i. At each moment of the simulation, we have $t_i \geq t, \forall i$. The algorithm repeats these steps until all tasks are marked executed:

- For each processor P_i,
  - we look at the next task to be executed on P_i (following the list scheduling given as input) if the current task is finished at time t;
  - if it is ready, we compute its full execution time by computing the time of reading the necessary input files, the weight of the task (given as input) and potentially some writing (in

case of crossover dependencies or if the checkpoint strategy requires this task to be checkpointed);
  - we look at the next error happening after time t: if it is before the end of the task then we set t_i to be the time of that failure, otherwise t_i is set to the time when the task ends and the task is marked executed.
- We set $t = \min_i t_i$.

There are two more things to detail: the computation of reading times and how we rollback when there is a failure. For the first problem, we keep a set of all files loaded on each processor. Before reading an input file, we check if it is already loaded (i.e., belongs to that set). If it is already loaded, we count a cost of 0, otherwise we add the communication/reading time for that file that is given as input. Files are added to the set whenever they are loaded or written (not necessarily a checkpoint). The set is cleared whenever a fail-stop error strikes on the processor.

When there is a failure, the rollback is easy because we always checkpoint crossover dependencies and the execution of a task always starts after the checkpoint is done. This implies that a failure on a processor P_i will only impact the tasks that have been executed on P_i since the last checkpointed task that was mapped to P_i. To rollback we explore the list of tasks backward from the current task to the last checkpointed one (we keep two pointers on these two tasks at each time to access them instantaneously), we mark each task unexecuted, we clear the set of loaded files and we can start simulating again from the last checkpointed task as if nothing happened. In the case of CKPTNONE, the simulation is rolled back from the first task anytime an execution or communication is interrupted.

Finally, the simulator computes the following measures: the number of file checkpoints taken, the number of task checkpoints taken, the number of failures, the total time spent checkpointing data and the execution time of the application.

3.3.3 Results

In this section, we first compare the expected makespan of our proposed checkpointing strategies (CDP and CIDP) over two baseline strategies (ALL and NONE) with the same task mapping and scheduling strategy. Remind that "C", "I" and "DP" stands for checkpointing crossover dependencies, checkpointing induced dependencies and dynamic programming respectively. Then, we compare the solutions (different task mapping and scheduling heuristics combined with several checkpointing strategies) from this work with the method PROPCKPT proposed in Chapter 2 for M-SPGS.

In Figure 3.6 through Figure 3.10, we compare the four considered task mapping and scheduling strategies: HEFT and MINMIN, with their chain-mapping variants HEFTC and MINMINC using boxplots. On these figures, the lower the better and the baseline at 1 is the performance of HEFT. The chain-mapping variants have the same performance or improve that of their basic counterparts, especially when communications are expensive (rightmost parts of the graphs). The other conclusion is that MINMIN (resp. MINMINC) almost always achieves same or worse performance than HEFT (resp. HEFTC). This is easily explained by the fact that HEFT and HEFTC take into account the critical path of workflows. These trends are representative of the trends that can be observed for all considered graphs and workflows, but suffer from some exceptions. The chain-mapping variants can be superceded by their basic counterparts for workflows that do not include any chains (like LU in Figure 3.7), because the basic variants can use backfilling. However, backfilling sometimes backfires, even in the absence of chains, like for SIPHT in Figure 3.9 where HEFTC can decrease the expected makespan by more than 30% with respect to HEFT. Overall, of the four considered task mapping and scheduling heuristics, HEFTC never achieves significantly bad performance, and most of the time achieves the best performance. This is the reason why we focus on it in the remainder of this section.

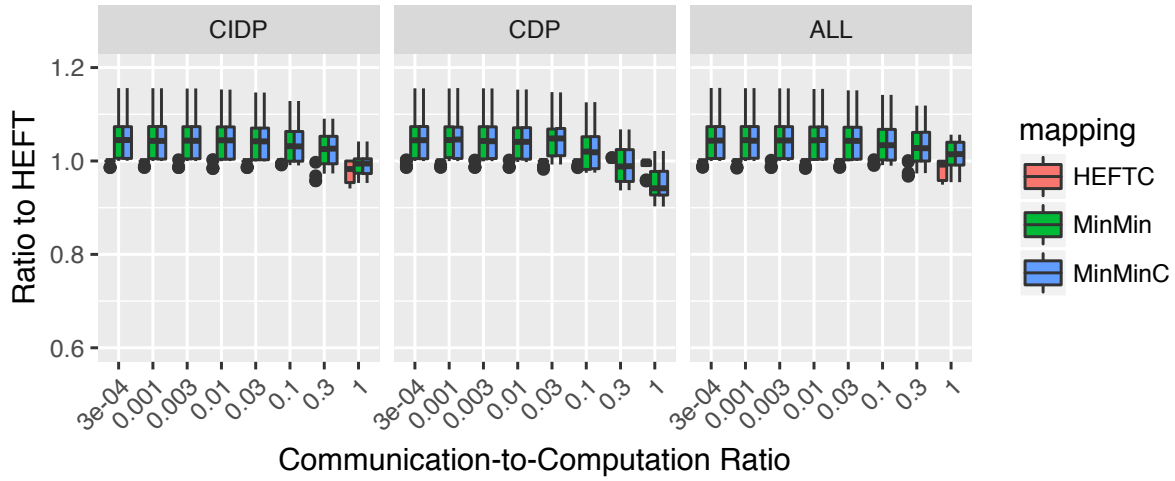


Figure 3.6: Relative performance of the four task mapping and scheduling strategies for Cholesky.

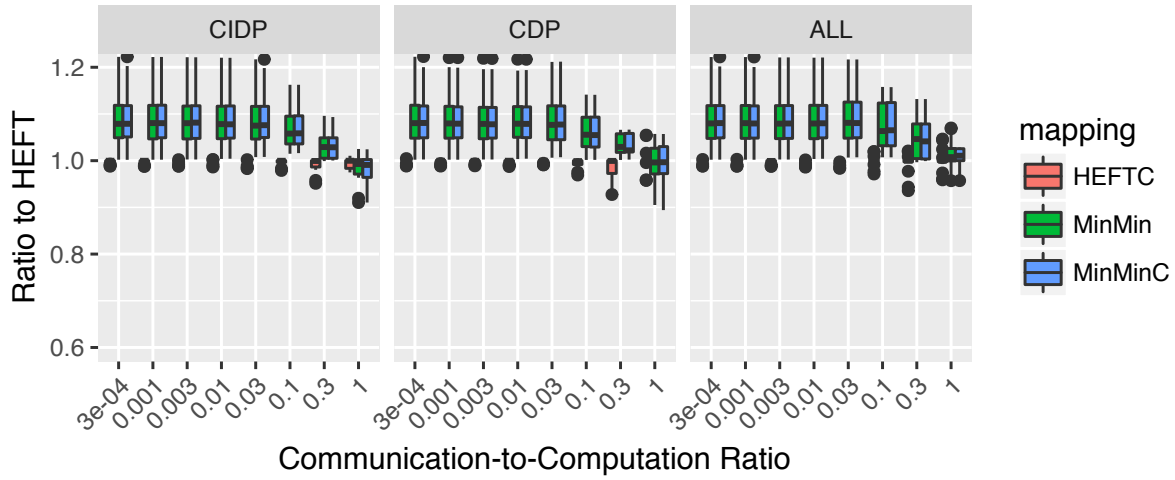


Figure 3.7: Relative performance of the four task mapping and scheduling strategies for LU.

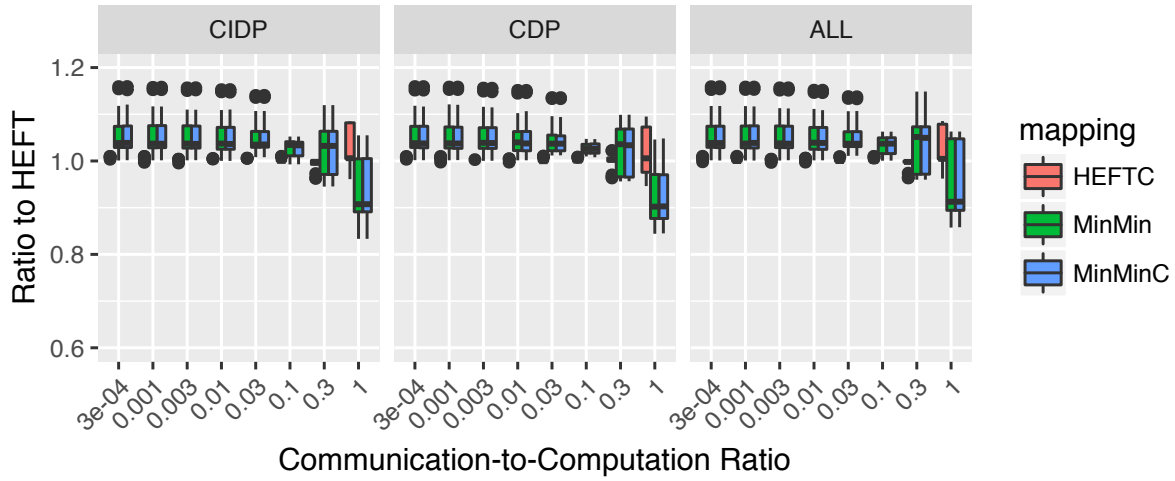


Figure 3.8: Relative performance of the four task mapping and scheduling strategies for QR.

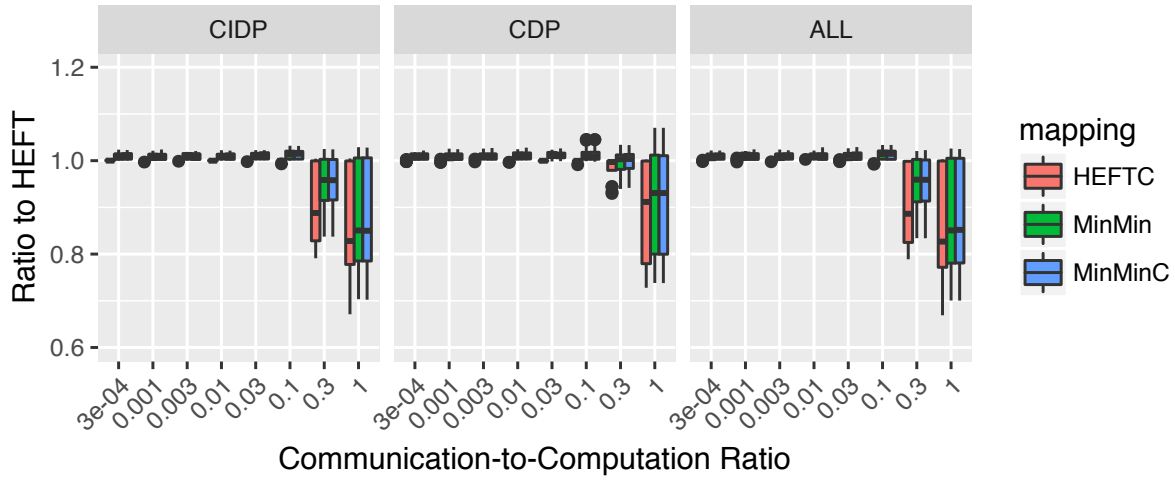


Figure 3.9: Relative performance of the four task mapping and scheduling strategies for Sipht.

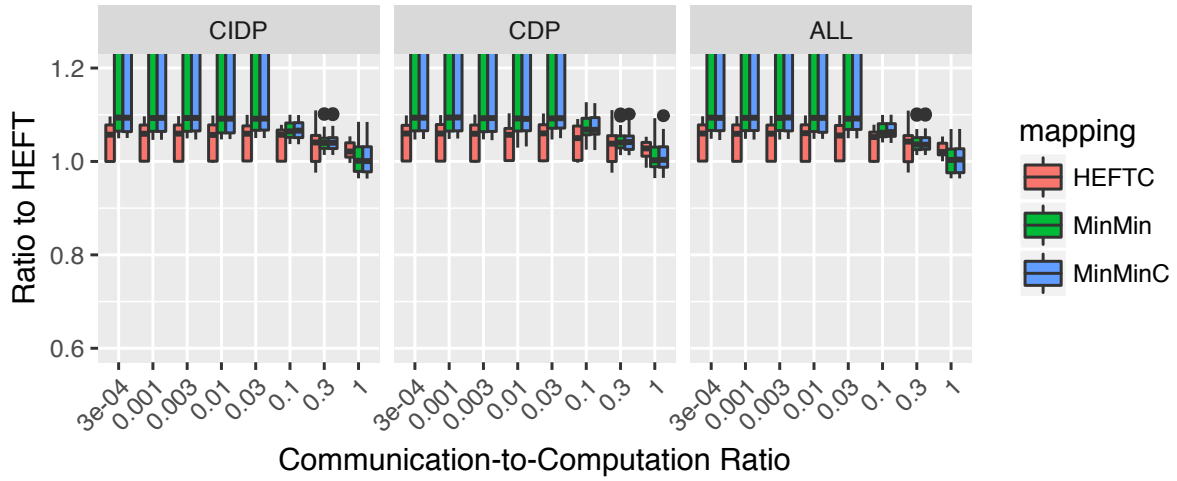


Figure 3.10: Relative performance of the four task mapping and scheduling strategies for CyberShake.

Figure 3.11 through Figure 3.18 present the expected makespans achieved by CDP, CIDP and NONE divided by that of ALL when the Communication-to-Computation Ratio increases. Therefore, the lower the better and data points below the $y = 1$ line denote cases in which these strategies outperform the competitor ALL (i.e., achieve a lower expected makespan). Each figure shows results for workflows with different number of tasks, ranging from 50 to 1240 tasks (each line of subfigure is for a different size, the number of tasks being reported on the rightmost column), for various number of processors P (different line styles), and for the three p_{fail} values (0.0001, 0.001, 0.01). We report on these figures the average number of failures that occur for the 10,000 random trials for each setting. These numbers are reported in black above the horizontal axis in each figure. The other two lines of numbers are the number of checkpointed tasks for the CDP and CIDP strategies, each number is printed with the same color as the curve of the corresponding strategy.

A clear observation is that CIDP never achieves worse performance than ALL: either it achieves a similar performance or it outperforms ALL, especially when communications, and thus checkpoints, are expensive (in the rightmost parts of graphs). It should be noted that when checkpoints come for free (leftmost parts of graphs), ALL and CIDP have the same performance as they do the same thing: they checkpoint all tasks. When the number of failures rises, the optimal solution is to checkpoint more tasks, potentially all of them, and the gain of CIDP with respect to ALL therefore decreases. This can be seen, for instance, on Figure 3.12 when $p_{\text{fail}} = 0.01$, $n = 385$ and there are 385 tasks checkpointed.

In the majority of cases, CDP also achieves similar or better performance than ALL. As we explained in Section 3.2, the dynamic programming algorithm is well-defined for CIDP, which checkpoints all induced dependencies. However, CDP tries to save some checkpointing overhead by not systematically checkpointing induced dependencies. As a consequence, the dynamic programming algorithm estimations of expected execution times may be inaccurate, which explains the sometimes bad performance of CDP. There are only a couple of CCR values for CYBERSHAKE for which CDP achieves a significantly worse performance than ALL. On the contrary, CDP often has better performance than CIDP when checkpointing cost is high. In all scenarios, CDP checkpoints less or the same number of tasks than CIDP. Depending on the checkpointing cost and failure rate, CDP can lead to significant improvement over ALL. For workflows as dense as LU, we save more than 10% when $CCR = 1$ for both strategies (see Figure 3.12), and CDP even achieves 35% saving for SIPHT (see Figure 3.17). As the CCR

Table I: Identify suitable cases to apply each approach

Strategy	Description
ALL	when checkpoints are free and/or extremely high p_{fail}
CIDP	always achieves a similar performance or outperforms ALL
CDP	better than CIDP when checkpointing cost is high
NONE	when checkpoints are expensive and/or extremely low p_{fail}

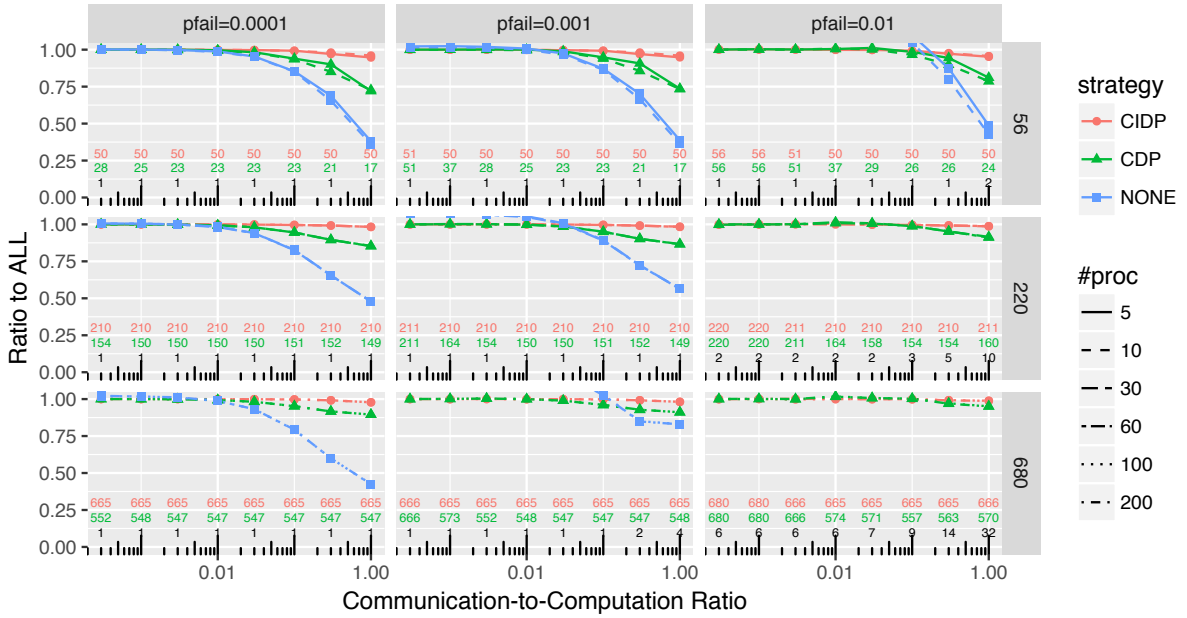


Figure 3.11: Performance of the different checkpointing strategies for Cholesky using HEFTC for task mapping and scheduling.

decreases, the ratio converges to 1. As already pointed out, this is because both strategies decide to checkpoint most, if not all, tasks, when checkpointing becomes cheaper.

CDP and CIDP achieve better results than NONE except when (i) checkpoints are expensive (high CCR) and/or (ii) failures are rare (low p_{fail}). In these cases, checkpointing is a losing proposition, and yet our strategies, by design, always checkpoints some files (they checkpoint all crossover files and even induced dependencies for CIDP). In practice, in such cases, the optimal approach is to bet that no failure will happen and to restart the whole workflow execution from scratch upon the very rare occurrence of a failure. NONE becomes worse whenever there are more failing tasks, i.e., when the failure rate increases (going from the leftmost column to the rightmost one in the figures), and/or when the number of tasks increases (going from the topmost row to the bottom one in the figures). When the failure rate is high and the workflows are large (the bottom right corner of the figures), the relative expected makespan of NONE is so high that it does not appear in the plots. The above results, and our experimental methodology in general, make it possible to identify these cases so as to select which approach to use in practical situations. We summarize the major observations in Table I:

Figure 3.19 presents the aggregated results for the 180 STG random DAGs with boxplots. The trends on these graphs are the same as already reported. This confirms the generality of our conclusions.

Finally, we compare our new general approach with PROPCKPT, the approach specific to M-SPGs that we proposed in Chapter 2. Figure 3.20 through Figure 3.22 present this comparison for Montage,

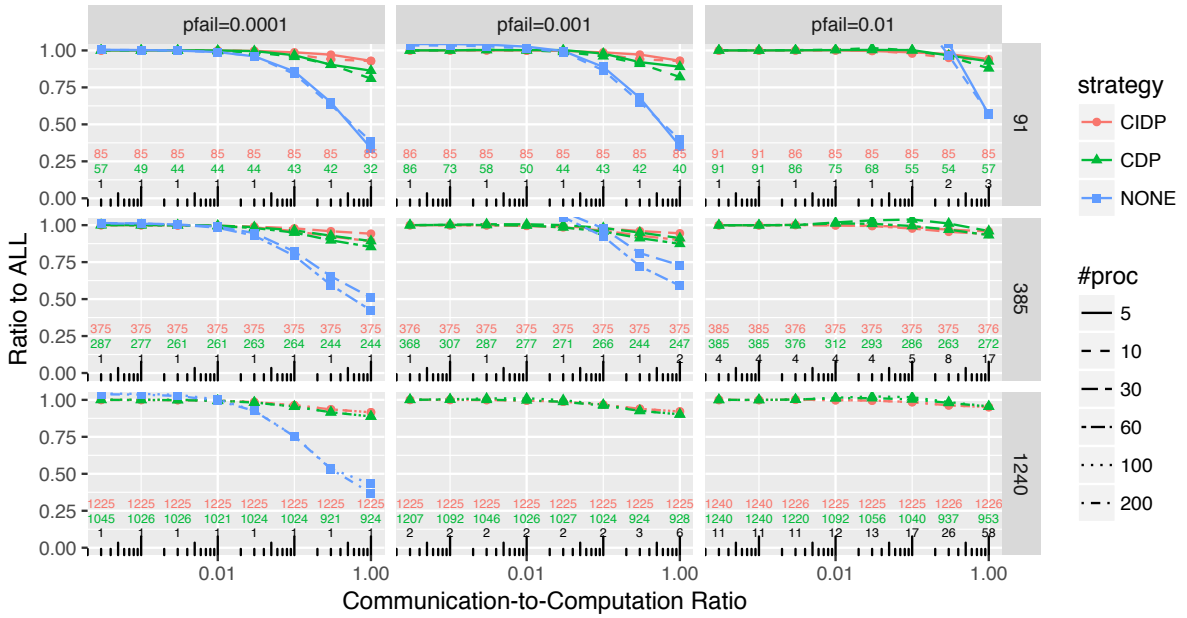


Figure 3.12: Performance of the different checkpointing strategies for LU using HEFTC for task mapping and scheduling.

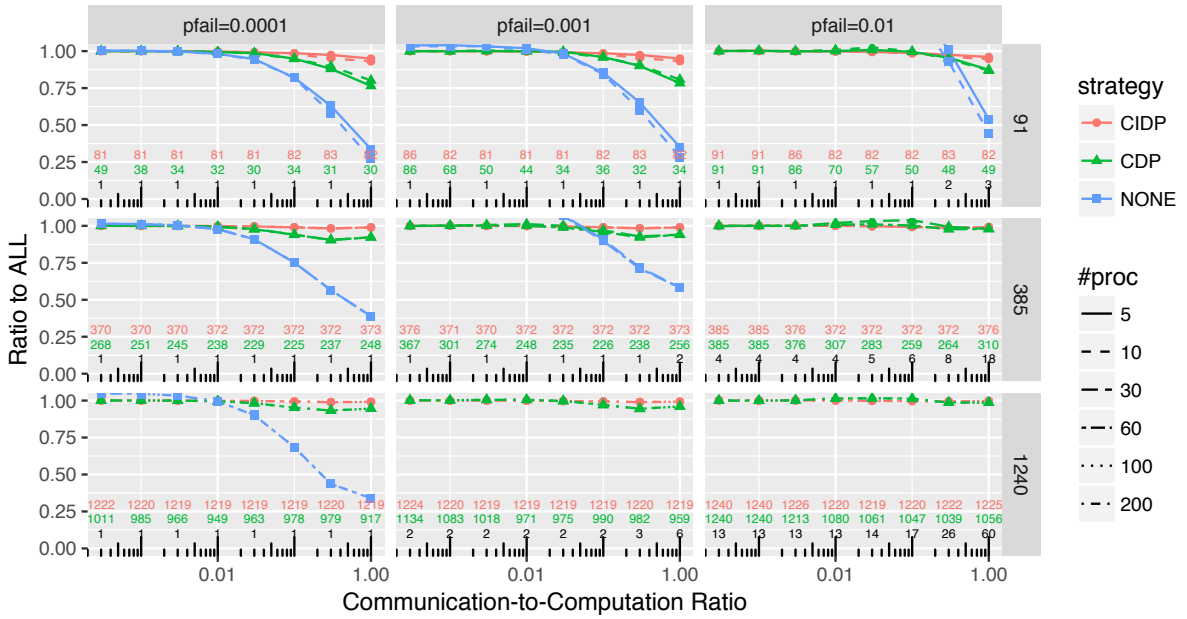


Figure 3.13: Performance of the different checkpointing strategies for QR using HEFTC for task mapping and scheduling.

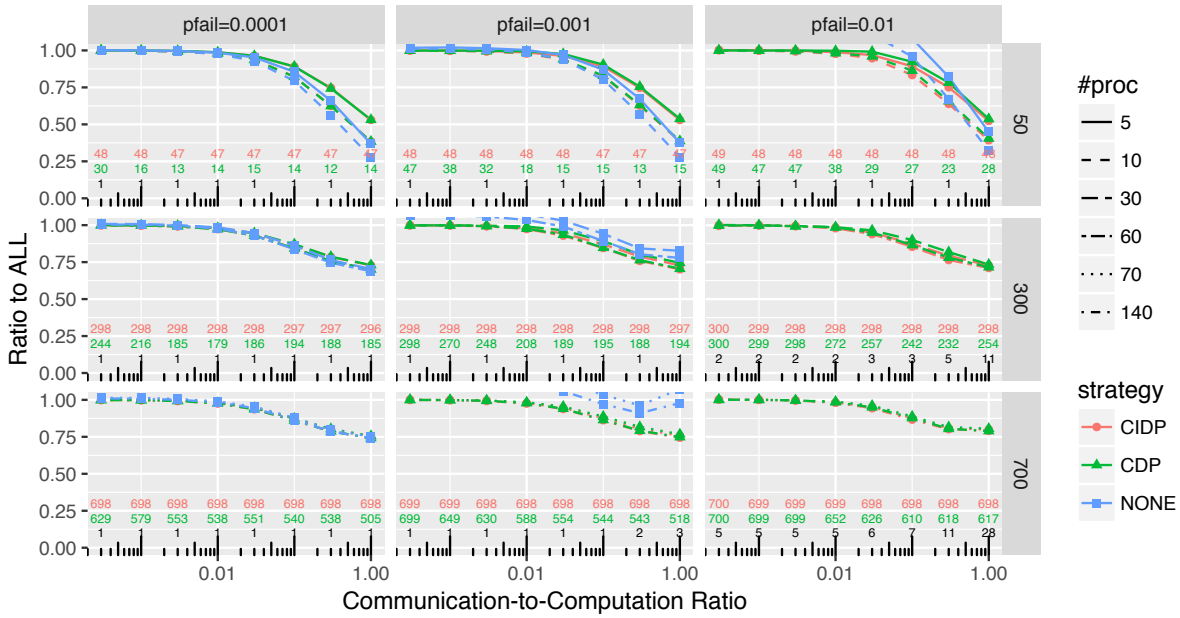


Figure 3.14: Performance of the different checkpointing strategies for Montage using HEFTC for task mapping and scheduling.

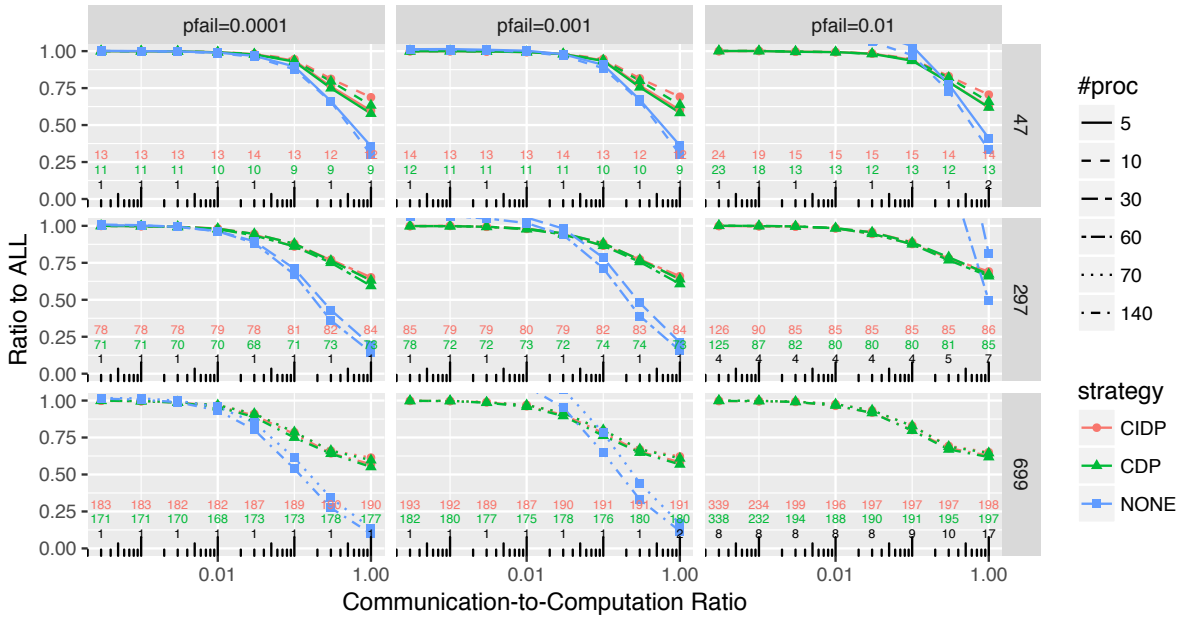


Figure 3.15: Performance of the different checkpointing strategies for Genome using HEFTC for task mapping and scheduling.

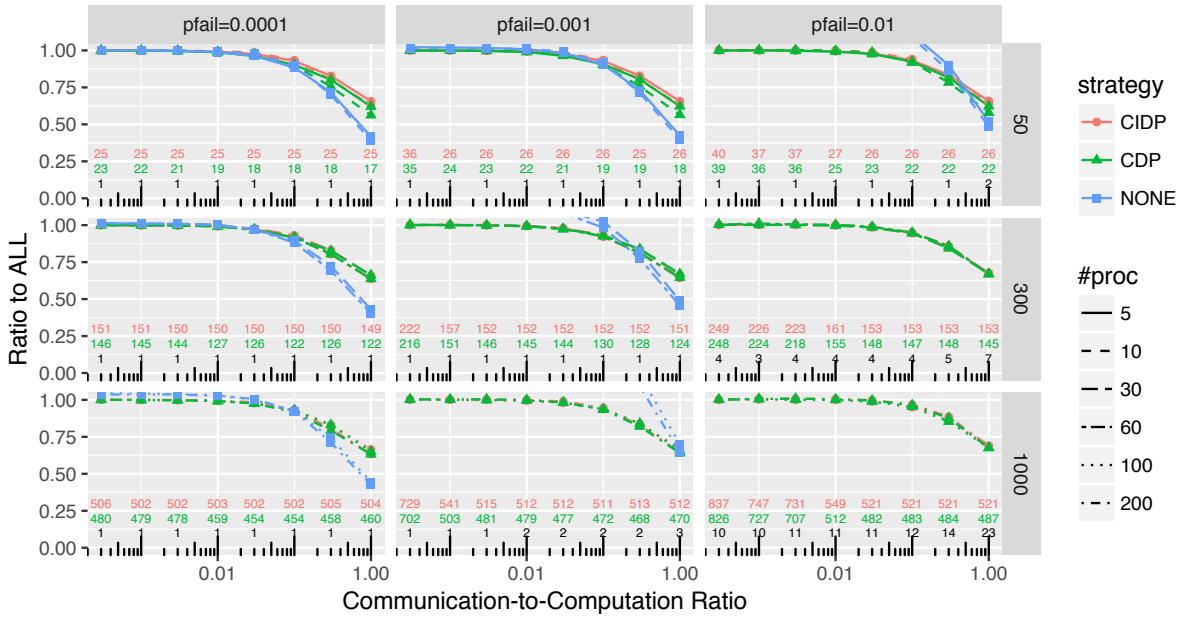


Figure 3.16: Performance of the different checkpointing strategies for Ligo using HEFTC for task mapping and scheduling.

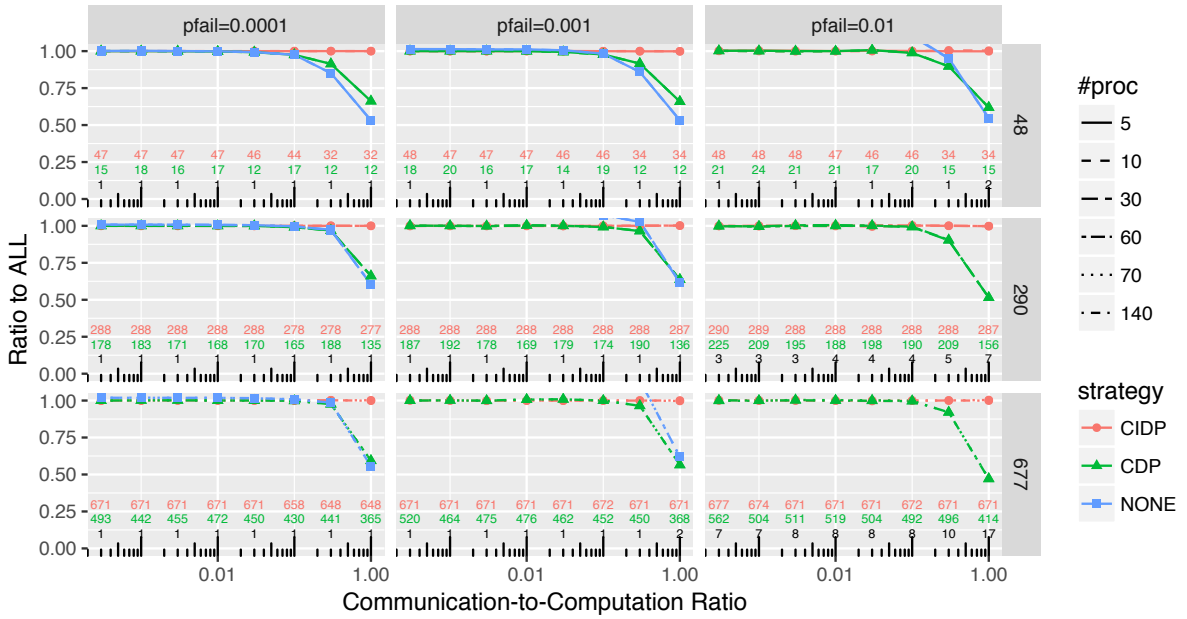


Figure 3.17: Performance of the different checkpointing strategies for Sipht using HEFTC for task mapping and scheduling.

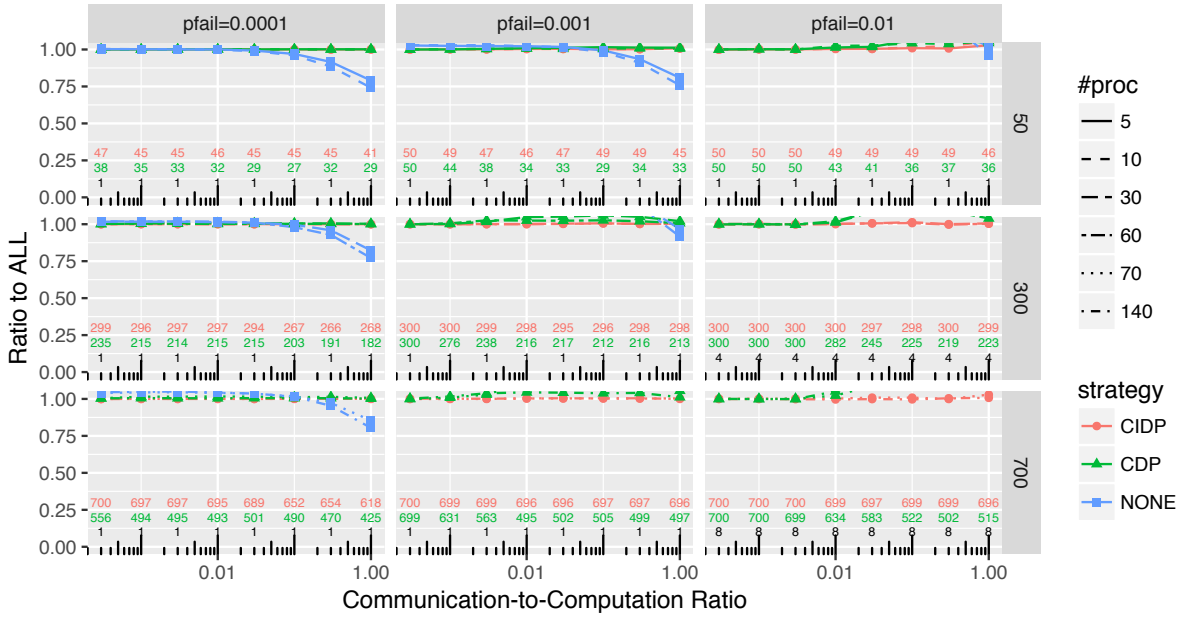


Figure 3.18: Performance of the different checkpointing strategies for CyberShake using HEFTC for task mapping and scheduling.

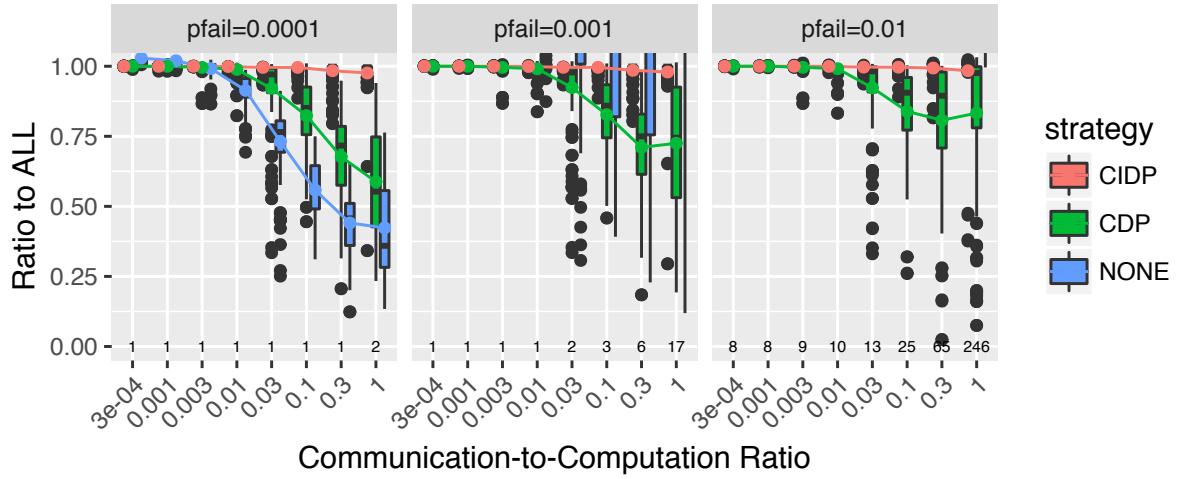


Figure 3.19: Average performance of the different checkpointing strategies for the STG task graphs using HEFTC for task mapping and scheduling.

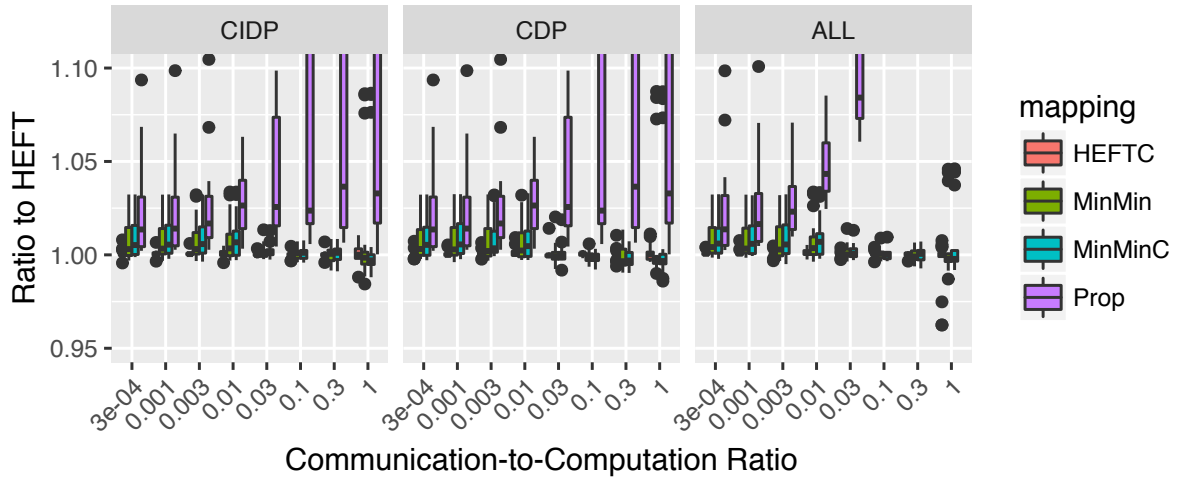


Figure 3.20: Relative performance of the four task mapping and scheduling strategies and of PROPCKPT for Montage.

Ligo and Genome, which are the three M-SPGS considered in Chapter 2. Overall, the new approaches perform better than PROPCKPT.

3.4 Conclusion

This chapter tackles the challenging problem of executing arbitrary workflows on homogeneous processors, with reasonable performance in presence of failures but without incurring a prohibitive cost when no failure strikes. While CKPTALL meets the first objective by expensively checkpointing every task and CKPTNONE meets the second one by avoiding any checkpoint at all, we propose new strategies that provide different trade-offs between these two extremes. First, all crossover dependencies, corresponding to file transfers between processors, are checkpointed, which prevents re-execution propagation between processors in case of failure. Then, a DP (Dynamic Programming) solution is used to insert additional checkpoints to minimize the expected completion time. Additional (induced) task checkpoints may be added prior to the DP execution to provide it with more accurate information. Moreover, different scheduling strategies that extend classical ones to reduce the number of checkpoints were also proposed. To the best of our knowledge, these new strategies are the first to be tuned to minimize the need for checkpointing while mapping tasks. Extensive experiments with a discrete event simulator, conducted for both synthetic and realistic instances, show that our approaches significantly outperform CKPTALL and CKPTNONE in most scenarios.

Future work will aim at extending our approach to workflows with parallel moldable tasks [36]. Such an extension raises yet another significant challenge: now the number of processors assigned to each task becomes a parameter to the proposed solutions, with a dramatic impact on both performance and resilience.

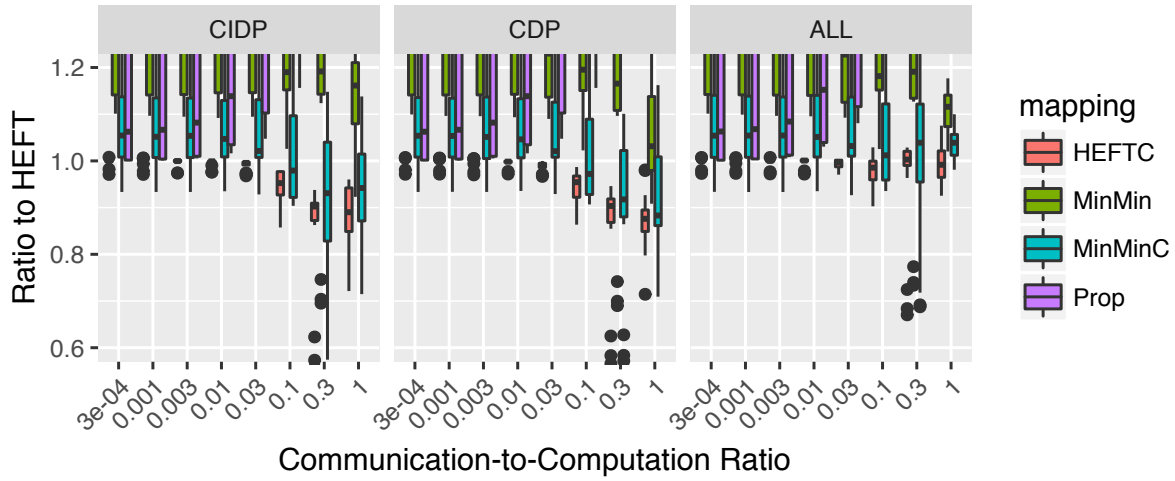


Figure 3.21: Relative performance of the four task mapping and scheduling strategies and of PROPCKPT for Ligo.

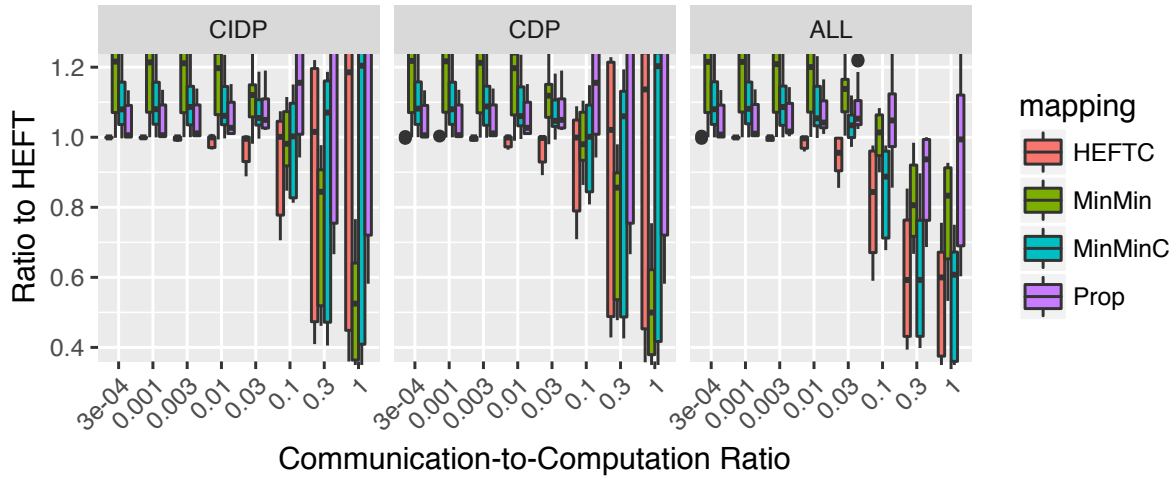


Figure 3.22: Relative performance of the four task mapping and scheduling strategies and of PROPCKPT for Genome.

Part II

Energy-aware strategies for reliability-oriented real-time task allocation

Chapter 4

Framework

4.1 Introduction

In the second part of this thesis, we focus on scheduling techniques to allocate periodic independent tasks onto real-time platforms subject to transient faults. In real-time systems, the workload consists of a specific set of periodic tasks with timing constraints, which is expressed as *deadlines*. The system must complete each task before its corresponding deadline. The timeliness of the output is as important as its logical correctness. This is because for real-time systems the lack of adherence to the predefined deadlines can (or will) result in catastrophic system failure. A periodic task has multiple instances that are released repetitively at well-determined time points. The minimum inter-arrival time between two consecutive instances is called the *period* of the task. A real-time task set typically contains multiple tasks, possibly with different periods. The least common multiple (LCM) of all the periodic tasks is called the *hyperperiod*.

A wide range of applications in real-time computing domain are safety-critical, e.g., railway control systems, aerospace system. Therefore, tolerating faults and achieving high reliability levels is a mandatory requirement. The most common way to tolerate transient faults in real-time system is replication, where several instances (replicas) of each task are executed by the system. In case of error, the primary replica of the task fails. However, we still have secondary replicas to ensure that we will achieve the predefined reliability target. Note that it is never possible to guarantee success in any case because of the probabilistic nature of failures.

Another very desirable feature of real-time systems is energy awareness. Replication strategy adds redundancy in system resources, e.g., spare processing units, to be able to execute more replicas of tasks. However, this conflicts with the energy-saving objective. If we do not consider energy, we can run all replicas at the maximum processor frequency and add as many replicas as possible to guarantee the reliability threshold of the system, but this may lead to a very large energy consumption. To be able to manage the energy consumption, we can use Dynamic Voltage and Frequency Scaling (DVFS) when executing tasks on processors. As stated in the Introduction section, DVFS is a well-known technique that trades off the processing speed with energy savings, but in the context of reliability, it has a negative impact on the transient fault rate: as we decrease the supply voltage and frequency to save power, the transient fault rate significantly increases [33]. As a consequence, when we scale down the frequency level, we must take into account the reliability degradation.

In a nutshell, this part deals with the following optimization problem: given a set of independent real-time tasks subject to (possibly different) periodic deadlines, how to execute them on a parallel platform and match all deadlines while minimizing the expected energy consumption? The problem is complicated by the need to enforce some reliability threshold.

In Chapter 5, first the focus is on homogeneous systems, which means that we have identical processing units. We revisit the most recent results on this problem presented by Haque, Aydin and Zhu [54]. We call [54] *the reference paper* throughout the text. In homogeneous systems, for each processor, we have a set of available frequency levels. For each task, we could calculate the number of replicas needed to meet the reliability goal at each frequency level before allocating it. The energy consumed by each single replica depends upon the frequency chosen for its execution. The lower the frequency, the less energy consumed. However, when a task has more than one replica, the total expected energy consumption (sum up of all replicas) does not decrease linearly with the frequency levels. The reason is that we need more replicas for a lower frequency level, and existing research indicates that arbitrarily slowing down a task is not always energy-efficient [39, 62, 123], due to the frequency-independent power. In other words, there is a processing frequency below which the total energy consumption increases. Moreover, a lower frequency leads to a longer execution time which may prevent the deadline to be matched. So we need to choose a frequency level for each task (or each replica) so as to minimize the energy consumption while satisfying all deadlines and the reliability threshold. Also, we noticed that it is sufficient to complete only one replica successfully. If the primary replica is successfully executed, then we can safely cancel the remaining replicas immediately in order to save energy. Consequently, if the execution of the replicas of a given task do not overlap, energy consumption will be minimal whenever the first execution succeeds. Based on this cancellation mechanism, we try to enhance each optimization step of the reference paper in order to reduce the overlapping between replicas, including how to choose the frequency settings, how to map the tasks to processors, and how to schedule them.

Then in Chapter 6, we extend these energy-aware strategies for allocating real-time tasks onto heterogeneous platforms. Heterogeneous platforms have been used for safety-critical systems for many years [44]. With the advent of multiple hardware resources such as multi-cores, GPUs, and FPGAs, modern computing platforms exhibit a high level of heterogeneity, and the trend is increasing. The multiplicity of hardware resources with very different characteristics in terms of speed profile, reliability level and energy cost, raises an interesting but challenging problem: given several device types, which ones should we keep and which ones should we discard in order to achieve the best possible tri-criteria trade-off (time, energy, reliability)? The design of mapping and scheduling heuristics is much more technical than in the case of homogeneous platforms (identical processors). Intuitively, this is because the reliability and energy consumption of a replica of a task instance depends upon the processor which executes it. We will explain the challenges induced by heterogeneity in more detail afterwards.

Overall, the objective is to minimize the expected energy consumption while matching all deadlines and reliability constraints. The expected energy consumption is the average energy consumed over all failure scenarios. Consider a sample execution: whenever the execution of a task replica succeeds, all the other replicas are instantaneously deleted; therefore, the actual amount of energy consumed depends both upon the error scenario (which replica is the first successful) and upon the overlap between replicas (some replicas are partially executed and interrupted when the successful one completes). Given a mapping, the scheduling should aim at minimizing the expected energy consumption by avoiding overlap between any two replicas of the same task, in other words, one should execute primary replica as soon as possible and postpone secondary replicas. Note that having an overlap-free scheduling is not always possible because of utilization constraints. Also, deciding whether an overlap-free scheduling exists for a given mapping is NP-hard, which will be proved in Section 5.3.5, Even if the task deadline is not constraining, the problem would remain NP-hard which will be proved in Section 6.1.5. Moreover, in actual real-time systems, tasks often complete before their worst-case execution times, or WCETs, so that execution times are routinely modeled as stochastic. In the end, the expected energy consumption must also be averaged over all possible values for execution times in addition to over all failure scenarios. Therefore, to assess the effectiveness of our heuristics, we will use a comprehensive set of execution

scenarios with a wide range of parameters. Besides comparing to the state-of-the-art strategy [54], which is available for the problem studied in Chapter 5, we analytically derive a lower bound for the expected energy consumption of any solution as a reference in Chapter 6 when there is no existing similar works.

4.2 Related work

We first review related work devoted to real-time task allocation on homogeneous platforms (Section 4.2.1). We then briefly summarize studies related to scheduling non-periodic tasks for heterogeneous platforms (Section 4.2.2). Finally, we review studies devoted to scheduling of real-time tasks on heterogeneous platforms (Section 4.2.3).

4.2.1 Scheduling real-time applications on homogeneous platforms

There is a very significant literature on real-time scheduling for multiprocessor systems. Liu and Layland first introduced the Earliest Deadline First (EDF) and the Rate Monotonic (RM) scheduling policies for real-time systems and provided the utilization bounds for both policies in 1973 [71]. Since then, the real-time scheduling problem has been extensively studied. However, most work is devoted to homogeneous processor systems, as exemplified by the survey [30] which ignores altogether heterogeneous systems, and by the more recent survey [91] where only 9 of the 78 references deal with heterogeneous platforms. [54] minimizes the energy when scheduling independent tasks with different deadlines on a homogeneous platform while satisfying some threshold on reliability. As we try to improve the solution from [54] in Chapter 5, here we review the relevant work citing paper [54] with the aim of complementing the related work already covered in the reference paper: in [98], Taherin et al. propose an approach for energy management that is only applied on "low-criticality tasks in low-criticality mode to preserve the original reliability of the system". This approach cannot guarantee that a reliability threshold is met. Keeping the maximum power consumption below the chip thermal design power, Ansari et al. [3] have proposed a peak power management approach to meet thermal design power in fault-tolerant system. However, in the scheduling task graph, all tasks have the same period/deadline. Cao et al. in [23] proposed an affinity-driven modeling and scheduling approach to makespan optimization. They optimize average peak temperature and makespan, but not energy minimization. To improve quality in real-time system, Cao et al. in [22] proposed QoS-adaptive approximate real-time computation optimization. Approximate results of tasks are allowed that each task is composed of a mandatory part and an optional one that refines the result of the mandatory task. However, failures are not considered. Zhou et al. consider in [121] both transient and permanent faults. They try to improve soft-error reliability while satisfying a lifetime reliability constraint, but do not attempt to minimize energy consumption. Moreover, [53] considers the same problem as [54]; however, it uses checkpointing to cope with failures when we consider replication.

We refer the interested reader to [30, 54, 91] for a comprehensive overview of the related work for homogeneous platforms. Heterogeneous platforms make the problem even harder because processors can have different speeds, energy costs, and failure rates. Therefore, the processor preferred for one task by one of the objectives and constraints —deadline satisfaction, energy minimization, reliability threshold satisfaction— may be the worst processor for another objective or constraint. Heuristics have thus to perform complicated trade-offs in our three-criteria settings.

4.2.2 Scheduling for heterogeneous platforms

There is a huge literature on scheduling for heterogeneous platforms, and even dedicated workshops. Here, we only refer some very recent work closely related to our problem but dealing with non-periodic tasks. [111] maximizes the reliability of an energy-constrained DAG executed on a heterogeneous platform while using DVFS. Conversely, [115] minimizes the energy consumption of a reliability-constrained DAG executed on a heterogeneous platform while using or not DVFS. A group of authors published a book [114] and several articles on the problem of DAG scheduling on heterogeneous platforms. In Chapter 2 of book [114] and in [112] these authors consider the energy minimization when scheduling a DAG with or without DVFS. In [113] they considered the same problem while satisfying some reliability goal. However, these two results do not consider reliability.

Overall, these studies do not consider real-time applications. The deadlines constraining real-time tasks and applications make problems significantly harder to tackle.

4.2.3 Scheduling real-time applications on heterogeneous platforms

Some related work targets the scheduling of real-time applications on heterogeneous platforms, but without considering fault tolerance. For instance, [118] targets the execution of a DAG, but considering neither energy consumption nor fault-tolerance (when DAGs are scheduled, tasks are always assumed to have the same deadline). [48] targets the execution of independent tasks that access shared resources, the access to resources being exclusive. Their objective is to maximize the number of instances for which a solution is found. [76], [86] and [117] minimize energy consumption by using DVFS, [76] when scheduling independent tasks, [86] a DAG, and [117] a moldable application. [105] considers the scheduling of independent tasks and DAGs under an energy constraint, while [103] considers the scheduling of independent tasks under a thermal constraint. [116] proposes a fully polynomial-time approximation scheme (FPTAS) for minimizing the energy consumption for a set of independent tasks executed on a set of heterogeneous (unrelated) processing elements.

Some of the related work considers the execution of real-time applications on heterogeneous failure-prone platforms but is limited to coping with a single failure per task or per processor. [82] maximizes the reliability of the considered DAG but does not consider energy consumption and follows the primary/backup technique and, thus, is limited to at most one failure per task of the DAG. [83] attempts to maximize resource utilization (and does not consider energy) when scheduling a set of independent tasks. It assumes that at most one processor can fail, which enables the simultaneous scheduling of several backup tasks on the very same processor as at most one of them will need to be executed. [58] minimizes the energy consumed for the execution of a DAG while satisfying a reliability threshold. The proposed solution uses DVFS and Power Mode Management (i.e., the ability to switch off idle processors to low-power inactive state). This solution, however, cannot produce a schedule more reliable than the original one. It also supports at most one fault per processor. [45] minimizes the energy consumed for the execution of a set of independent tasks while satisfying a reliability threshold using DVFS and following a primary-backup approach.

Very few studies consider the execution of real-time applications on heterogeneous failure-prone platforms and can cope with two or more failures per task. [95] minimizes the energy consumed for the execution of a set of independent tasks while satisfying a reliability threshold. The proposed solution uses DVFS. This solution, however, is based on a primary-backup approach that is then extended. This approach, by design, cannot produce a schedule more reliable than the original one with two replicas per task, strongly relies on DVFS, and schedules several replicas of a same task on the same processor (what most other approaches forbid). [44] targets the execution of a DAG on a heterogeneous platform while satisfying a reliability threshold. However, the objective is not the minimization of energy consumption

but the maximization of the *utilization of energy consumption*, which can be seen as a yield of reliability improvement with respect to increase energy consumption. As a consequence, [44] produces energy greedy schedules (see subplots (a-1), (b-1), and (c-1) of Figure 1 in [44]). In Chapter 3 of the already mentioned book [114], the authors consider cost minimization (which can be energy minimization) when scheduling a DAG under deadline and reliability constraints. Therefore, we consider the same problem but for a set of independent tasks rather than for a DAG. Because of the dependence between tasks and the chosen as-soon-as-possible scheduling of [114], this solution tends to schedule simultaneously the different replicas of a single task. As discussed previously that this can lead to a significant waste of energy. Therefore, it would have been unfair to compare our solution to that of [114] applied on independent tasks.

From what precedes, we have only identified a single existing solution that enables to schedule real-time tasks on heterogeneous platforms while minimizing energy consumption and satisfying some bound on the overall reliability as the problem will be studied in Chapter 6. However, this solution being dedicated to DAGs lacks the possibility to minimize the overlapping between replicas of a same task, what we specifically targeted (cf. Section 6.3).

Chapter 5

Homogeneous platforms

In this chapter, we study the optimization problem starting with homogeneous platforms, and revisit the most recent results solving this problem presented by Haque, Aydin and Zhu [54] (the reference paper). Section 5.1 provides a detailed description of the optimization problem, and of the three-step approach used to solve it: (i) for each task, compute a set of replicas and their frequencies; (ii) map and statically schedule all replicas onto the platform; (iii) dynamically update the schedule based on actual completion times (instead of worst-case ones) and observed successful executions. These three steps are described in Section 5.1.2 to Section 5.1.4 respectively. We try to improve this state-of-the-art approach, in terms of energy saving, by refining it step by step. The main contributions of this chapter are the following:

- An example to identify the limitation of the scheduling strategies proposed in [54] (Section 5.2). Then we improve the latest approach in the reference paper through three steps listed below;
- In the first step, we revisit the estimation of energy consumption and use a different formula, which is expected to be closer to the actual execution scenarios (Section 5.3.1);
- In the mapping phase, we use a layered Worst-Fit Decreasing heuristic to avoid imbalance between processors, and try to evenly allocate the primary replicas of different tasks (Section 5.3.2);
- For the dynamic schedule, we implement several novel aggressive strategies to reorder and delay chunks in order to reduce overlapping between replicas of the same task. The basic ideas are described in Section 5.3.3 and Section 5.3.4 summarizes these new heuristics;
- A new complexity result that establishes the combinatorial nature of the scheduling step when the mapping is given (Section 5.3.5);
- A comprehensive experimental comparison of the results of the reference paper against those obtained with our improved approach in Section 5.4.[1]

In addition to the above sections, Section 5.5 gives concluding remarks and hints for future work.

5.1 Previous approach

In this section, we present the optimization problem in full details, and we describe the approach of the reference paper. Key notations are summarized in Table I.

5.1.1 Optimization problem

The inputs to the optimization problem are a set of real-time tasks, a set of processors and a reliability target:

[1] The algorithms are implemented in C++ and R and the related code, data and analysis are available at https://doi.org/10.6084/m9.figshare.9778319.v1.

Tasks – There are N periodic real-time tasks $\tau_1, \tau_2, ..., \tau_N$. Task τ_i has worst-case execution time (WCET) c_i under the maximum available frequency f_{max}. Tasks actually complete execution earlier than their estimated WCET: execution times are assumed to be data-dependent and non-deterministic, randomly sampled from some probability distribution whose support is upper bounded by the WCET. Task τ_i generates a sequence of *instances* with period p_i, which is equal to its deadline. The whole input pattern repeats every hyperperiod of length $L = \text{lcm}_{1 \leq i \leq n} p_i$. Each task τ_i has $\frac{L}{p_i}$ instances within the hyperperiod.

Processors – The platform consists of M homogeneous processors, with same set F of frequencies ranging from f_{min} to f_{max}. Without loss of generality, we normalize the frequencies to enforce $f_{max} = 1$. At frequency f_j, a processor needs up to c_{i,f_j} seconds, which is equal to $\frac{c_i}{f_j}$ in the reference paper, to complete an instance of task τ_i. The *utilization* u_{i,f_j} of task τ_i at frequency f_j is defined as $u_{i,f_j} = \frac{c_{i,f_j}}{p_i}$. The utilization of a processor is the sum of the utilizations of all tasks that are assigned to it.

Fault model – One considers transient faults, modeled by an exponential distribution with average arrival rate λ. It has been widely proved that the fault rate λ increases when the frequency is scaled down to save energy using DVFS [33]. Let λ_0 denote the fault rate at frequency f_{max}. Then the fault rate at frequency f_i is $\lambda(f_i) = \lambda_0 \times \exp^{\frac{d(1-f_i)}{1-f_{min}}}$, where d is the sensitivity factor; d is a measure of how quickly the transient fault rate increases when the system supply voltage and frequency are scaled. At the end of execution, there is an *acceptance test* to check the occurrence of soft errors induced by the transient faults. It is assumed that acceptance tests are 100% accurate. The duration of the test is included within the task WCET. The *reliability* of a task instance is the probability of executing it successfully, in the absence of permanent faults. The reliability of a single instance of task τ_i running at frequency f_j is $R_i(f_j) = \exp^{-\lambda(f_j) c_{i,f_j}}$.

Reliability threshold – Let $\mathcal{R}_i$ denote the reliability threshold for each instance of task τ_i; $\mathcal{R}_i$ may be given as part of the input as in Chapter 6, or the reliability threshold is given for the whole task system over the hyperperiod as in the reference paper: then $\mathcal{R}_i$ is computed using the *Uniform Reliability Scaling* technique [120]. We have $\omega = \frac{\phi_{i,target}}{\hat{\phi}_i}$ for all i, where ω is the uniform probability of failure scaling factor (given as part of the input), $\phi_{i,target}$ is the failure probability of task τ_i ($\phi_{i,target} = 1 - \mathcal{R}_i{}^{h_i}$, where h_i is the number of instances of task τ_i in the hyperperiod) and $\hat{\phi}_i$ is the failure probability of task τ_i when a single replica at maximum frequency is executed ($\hat{\phi}_i = 1 - R_i(f_{max})^{h_i}$). This leads to $\mathcal{R}_i = \sqrt[h_i]{1 - \omega(1 - R_i(f_{max})^{h_i})}$. Now, given the reliability threshold $\mathcal{R}_i$, the question is to determine how many replicas to use, and at which frequency to execute them, so that $\mathcal{R}_i$ is enforced while energy consumption is kept minimal. Note that all replicas of a given task instance will have the same execution time if run at the same frequency, because they operate on the same data.

Optimization objective – The objective is to determine a set of replicas for each task and their execution frequencies, and to build a static schedule of length the hyperperiod, where the replicas of each instance of each task are mapped onto the processors, so that energy consumption is minimized, while matching the deadline p_i and reliability threshold $\mathcal{R}_i$ for each instance of each task τ_i. We detail below how energy consumption is estimated. To further complicate matters, the static schedule is dynamically modified on the fly to take actual execution times rather than WCET into account. Also, as soon as one replica of a given task instance completes its execution successfully, all its other replicas become redundant and are terminated instantaneously.

Table I: Key Notations

Notation	Explanation
N and M	number of tasks and of processors
c_i	WCET for task τ_i under max. available frequency
p_i	period (deadline) for each task instance of task τ_i
h_i	number of instances of task τ_i in the hyperperiod
f_j	frequency $f_j \in F = \{f_{min}, \ldots, f_{max} = 1\}$
c_{i,f_j}	WCET for task τ_i at frequency f_j
u_{i,f_j}	utilization of task τ_i at frequency f_j
$f_{opt(i)}$	most energy-efficient frequency for task τ_i
$R_i(f_j)$	reliability of one instance of task τ_i under f_j
$\mathcal{R}_i$	target reliability threshold for one instance of task τ_i
w	uniform probability of failure scaling factor
k_{i,f_j}	min. replica number for task τ_i under f_j to meet $\mathcal{R}_i$
$E(\tau_i, f_j, k_{i,f_j})$	energy cost for task τ_i with k_{i,f_j} replicas under f_j

5.1.2 Replica sets

In the reference paper, the first step is to construct a table with all information needed. For each task τ_i and each possible frequency we compute the number of replicas needed, the corresponding energy cost and the CPU time.

Given a frequency f_j, we start by computing the number k_{i,f_j} of copies that are needed for (each instance of) τ_i. The reliability using a single task instance (no replica) is $R_i(f_j) = \exp^{-\lambda(f_j)c_{i,f_j}}$. If $R_i(f_j) \geq \mathcal{R}_i$, the reliability threshold is enforced, and we need no replica, hence $k_{i,f_j} = 1$. Otherwise, using r additional replicas, the reliability increases to $R_i^r(f_j) = 1 - (1 - R_i(f_j))^{r+1}$ (the task fails only if all $r + 1$ copies fail), and we take the minimum value r such that $R_i^r(f_j) \geq \mathcal{R}_i$. This leads to $k_{i,f_j} = r + 1 = \left\lceil \frac{\log(1-\mathcal{R}_i)}{\log(1-R_i(f_j))} \right\rceil$ with $\mathcal{R}_i = \sqrt[h_i]{1 - \omega(1 - R_i(f_{max})^{h_i})}$ (see Section 5.1.1). The reference paper maps different copies onto different processors, so necessarily $k_{i,f_j} \leq M$. If no value of k_{i,f_j} can be found, frequency f_j cannot be used, and a higher frequency must be selected.

Once we have determined the number k_{i,f_j} of copies of τ_i at frequency f_j (with $1 \leq k_{i,f_j} \leq M$), we compute the corresponding energy cost. The reference paper adopts a conservative strategy and sums up the energy cost of all copies. This is pessimistic because as soon as a copy is successful, the remaining copies are interrupted (if already started) or simply cancelled (if not started). The energy cost of a copy is estimated as the power times the execution time $c_{i,f_j} = \frac{c_i}{f_j}$, which is an upper bound. As for the power $P(f_j)$ at frequency f_j, we use

$$P(f_j) = P_{static} + gP_{dyn}(f_j) = P_{static} + g(P_{indep} + C \times f_j^3)$$

where P_{static} (the static power), P_{indep} (the frequency-independent part of dynamic power) and C (the effective switching capacitance) are system-dependent constants. g represents the system state and indicates whether dynamic power $P_{dyn}(f_j)$ is currently being consumed or not: when executing a task at frequency f_j on the processor, $g = 1$, otherwise $g = 0$. The energy cost $E(\tau_i, f_j, 1)$ for one copy of task τ_i at f_j is then: $E(\tau_i, f_j, 1) = P(f_j) \times c_{i,f_j}$. The final energy cost with k_{i,f_j} copies is estimated as $E(\tau_i, f_j, k_{i,f_j}) = k_{i,f_j} \times E(\tau_i, f_j, 1)$. The total CPU time is then estimated as $S(\tau_i, f_j, k_{i,f_j}) = k_{i,f_j} \times c_{i,f_j}$.

Furthermore, each processor always consumes static power when idle (this consumption can be eliminated only by a complete shutdown). Hence, we account for static power whenever the mapping and scheduling phases described below leave processors idle.

5.1.3 Mapping and static schedule

The first step provides an initial *configuration* as input to the second step, the mapping and static scheduling onto processors. The initial configuration consists of an assigned frequency and number of replicas for each task instance. Given a configuration, the mapping builds a schedule for an hyperperiod of length $L = \text{lcm}(p_i)$ as follows:
• sort the tasks by decreasing total CPU time; renumber them to have

$$S(\tau_1, f_{j(1)}, k_{1,f_{opt(1)}}) \geq S(\tau_2, f_{j(2)}, k_{2,f_{opt(2)}}) \geq \dots$$

where $f_{opt(i)}$ is the frequency leading to the lowest energy consumption to meet the reliability threshold;
• for i ranging from 1 to N, successively map all $k_{i,f_{opt(i)}}$ copies of τ_i onto $k_{i,f_{opt(i)}}$ different processors, using the First-Fit Decreasing (FFD) bin packing heuristic [64]. When mapping all the $\frac{L}{p_i}$ instances of a given task copy on a processor (bin), we use the standard *Earliest Deadline First (EDF)* scheduling heuristic [71]. EDF tells us that a given processor (bin) is a fit for that copy if and only if the utilization of that processor does not exceed 1. Recall that the utilization of a processor is the sum of the utilizations of all task instances assigned to it.

Hence, for a given task, the mapping finds the first processor whose utilization makes it a fit for the first task copy (and all its instances). Then it finds the first next processor whose utilization makes it a fit for the second task copy (and all its instances), and so on. If the mapping succeeds, we have built a static schedule for the hyperperiod. But it may well be the case that it is impossible to find a processor onto which to map a given task copy in the procedure, because all processor utilizations are too high to accommodate that copy. Then the reference paper proposes to enter an iterative procedure as follows:

- Change the initial configuration into the one where every task copy executes at maximal speed $f_{max} = 1$. This requires to fetch the values $k_{i,f_{max}}$ from the table and reordering the tasks by decreasing total CPU time $S(\tau_i, f_{max}, k_{i,f_{max}})$.
- Apply the mapping heuristic (FFD mapping and EDF schedule) to the new configuration.

Now, if the latter mapping fails again, there is no solution, resource utilization is too high. However, if the mapping succeeds, its energy cost may be very high. The reference paper proposes a refinement scheme where some tasks are *relaxed*, meaning that their frequency is decreased down to its predecessor (going from their current value f_j down to f_{j-1}). Initially, all tasks have frequency f_{max}, and some tasks are greedily selected for relaxation. If relaxing a task τ_i fails to lead to a successful mapping, or if we have reached its energy-optimal frequency $f_{opt(i)}$, then τ_i is marked as *ineligible*. The scheme stops when all tasks become ineligible. The reference paper uses three different greedy criteria to pick up the next task to be relaxed among eligible tasks, *Largest Energy First (LEF)*, *Largest Power First (LPF)*, and *Largest Utilization First (LUF)*. We refer to the reference paper for details. We have implemented the first two variants, LEF and LPF, because they are shown to outperform LUF in the reference paper.

5.1.4 Dynamic schedule

The static schedule, also called *canonical* schedule, is based upon the WCET of each task and EDF. It is never executed exactly as such, because the actual execution time of a task instance will be shorter than its WCET. Still, it is used as the baseline to guide dynamic updates. From the canonical schedule, each processor has an *assignment list* made up with all task instances that it has to execute during the hyperperiod.

Let us follow the operation of a given processor P. For simplicity, assume that P computes the full EDF schedule for all tasks in its assignment list, during the entire hyperperiod. The reference paper uses a data structure, called Canonical Execution Queue (CEQ), to avoid the high cost of computing the static

schedule, while preserving the same outcome. Recall that the EDF schedule uses the WCET of each task, and preemption, so that a given task instance may well be split into several chunks. Hence P has an ordered list of chunks together with their starting and finish times in its EDF schedule.

Let t be the starting time of the next chunk ch, from task instance τ, to be processed by P in its EDF schedule (initially, $t = 0$). In the canonical schedule, this chunk executes in the interval $[t_s, t_f]$, where t_s is the starting time and t_f the finish time on P. Let k be the number of copies of τ in the canonical schedule, distributed over k different processors. A major idea of the reference paper is to differentiate the action of P depending upon whether its own copy of τ is the first copy to start execution among the k processors. So if the chunk ch is indeed the first chunk of any copy of τ to start execution, then P promotes its copy of τ to the status of *primary* replica, and all the chunks of τ as primary chunks. P signals the other $k - 1$ processors that their copies (and their copy chunks) are *secondary* replicas (or chunks). Otherwise, the chunk ch has already been marked either as primary (if it is not the first chunk of τ, but τ has been marked primary on P previously) or as secondary (if some other processor has signalled P previously). The action of P is the following:

- if ch is a primary chunk, then P starts its execution immediately, using the frequency given by the canonical schedule. This execution will last for a duration of $t_f - t_s$ seconds in the worst case;
- if ch is a secondary replica, then P executes it at frequency f_{max}, and using ALAP (As Late As Possible) scheduling, as further detailed below.

The rationale for executing secondary chunks at highest frequency f_{max} is that it allows for a minimal execution time, hence a maximal delay for ALAP scheduling. When delaying secondary chunks, we hope that the primary copy will complete before secondary copies actually start, hence will be cancelled whenever the primary copy succeeds. At least, this ALAP strategy should minimize overlap between primary and secondary copies, hence minimize redundant work. We point out that the choice for primary/secondary replicas is done dynamically, by the first processor to start a task instance in the actual execution of the hyperperiod. It may well be the case that two different instances of the same task have not the same primary processor.

There remains to explain in full details how secondary chunks are scheduled on P. In fact, all secondary chunks are delayed according to the finish time of τ in the canonical schedule. Recall that the canonical schedule provides an execution interval $[t_s, t_f]$ for every chunk of τ. Assume there are m chunks. For notational convenience, let $[t_s(i), t_f(i)]$, $1 \leq i \leq m$ denote the m execution intervals in the canonical schedule. The sum of these m interval lengths $t_f(i) - t_s(i)$, $1 \leq i \leq m$ is equal to $\frac{c}{f}$, where c is the WCET of τ and f is its frequency from the canonical schedule. Since we have decided to execute τ at frequency $f_{max} = 1$, we only need c seconds, in the worst-case, to execute τ. For instance, if $f = 0.5$, we only need (at most) half the time planned in the canonical schedule. The reference paper uses backfilling and reserves a total of c seconds in the dynamic schedule, starting from the last interval and going backwards to the first interval, allocating slots until c seconds are reserved. Then P will execute τ greedily using these intervals from the beginning until completion of τ. Finally, all chunks of τ will be scheduled across n new intervals $[t'_s(i), t'_f(i)]$, where $1 \leq i \leq n$ and $n \leq m$, because very likely τ will finish before its WCET. Here are two examples with $c = 20$, $f = 0.5$ and $m = 3$:

- the intervals in the canonical schedule are $[5, 35]$, $[40, 46]$ and $[50, 54]$. We reserve 20 seconds out of the 40 available in these three intervals by keeping the third interval entirely (4 seconds), then keeping the second interval entirely (6 seconds) and then keeping the last 10 seconds of the first interval. Then P will use these reserved slots ($[25, 35]$, $[40, 46]$ and $[50, 54]$) to execute its copy of τ at frequency f_{max} starting from time 25.
- the intervals in the canonical schedule are $[10, 18]$, $[40, 46]$ and $[50, 76]$. We reserve 20 seconds out of the 40 available in the three intervals by just keeping the last fraction $[56, 76]$ of the third interval (20 seconds), and leaving the first and second intervals empty.

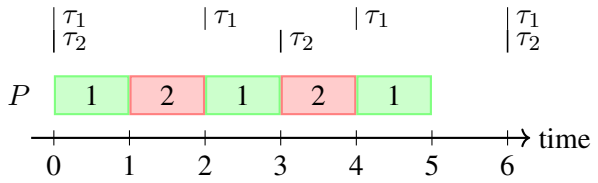


Figure 5.1: Canonical Execution Queue (CEQ) with two tasks τ_1 and τ_2 ($c_1 = c_2 = 0.5$, $p_1 = 2$ and $p_2 = 3$) on each processor, at frequency $f = 0.5$.

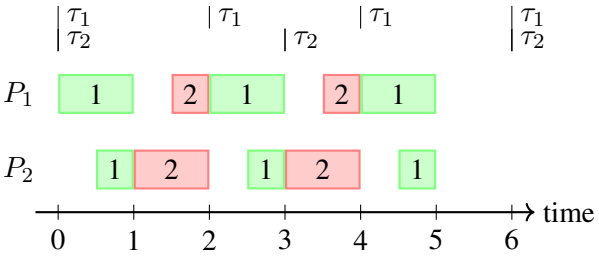


Figure 5.2: Executions when prioritizing primaries while delaying secondaries at maximum frequency as in the Canonical Execution Queue (CEQ).

In both cases, P will consume the slots from the beginning of the first reserved interval, until consuming all time units needed to finish τ, say at time t' ($t' \leq t'_f(n)$). Once the copy of τ on the processor P is successfully executed, all other copies (chunks) of τ are removed from the EDF list on other processors. Then P will start processing the next chunk ch' in its list right at time t' if ch' is primary, otherwise, the chunks are delayed using the same mechanism.

To summarize, consecutive primary replicas are scheduled ASAP (As Soon As Possible), while secondary replicas are scheduled ALAP, which aims at reducing the overlap between copies over different processors, so as to minimize energy consumption.

5.2 Motivational example

We illustrate one limitation of the strategy in [54] with a simple example with two tasks τ_1 and τ_2 and two processors P_1 and P_2. Assume that: (i) for WCET, $c_1 = c_2 = 0.5$ at maximum frequency $f_{max} = 1$; (ii) for periods $p_1 = 2$ and $p_2 = 3$; (iii) τ_1 and τ_2 have been scheduled to run at frequency $f = 0.5$, which takes 1 time unit; and (iv) the reliability threshold is met with one replica per processor for each task. For simplicity, also assume that: (i) each instance of τ_1 is a primary replica on P_1 and secondary on P_2, while each instance of τ_2 is a primary replica on P_2 and secondary on P_1; and (ii) there is no difference between actual execution times and the WCETs.

Figure 5.1 shows the CEQ (same for both processors) for the example. Figure 5.2 shows the execution on both processors with a CEQ-based strategy as described in Section 5.1.4. Recall that for secondary replicas, the strategy delays their executions according to CEQ and runs them at f_{max}. In the example, there is a complete overlap between each primary and each secondary replica due to the constraint given by the CEQ. This is wasteful because the secondary replica can never be canceled whenever the primary replica succeeds. However, it would be possible to avoid any overlapping by advancing each primary replica while delaying each secondary replica. In the example, this simply means interchanging

τ_1 and τ_2 on P_2. Without overlap, the executions of all secondary replicas at maximum frequency will be cancelled most of the time, leading to a substantial energy gain.

5.3 New strategies

We identify several possible reasons why the approach in the reference paper may be sub-optimal:

- All optimizations in the dynamic schedule aim at reducing overlap among replicas, so as to avoid redundant work. In their final schedule, the primary replica of task τ_i is executed at assigned frequency f_j, but the frequency of all $k_{i,f_j} - 1$ secondary replicas is increased to f_{max}. However, energy consumption is estimated with all replicas at frequency f_j. We revisit the estimate for energy consumption and use a different formula, with one copy executing at f_j, and the rest at f_{max}: this is expected to be closer to actual execution scenarios;
- The mapping uses the *First-Fit Decreasing (FFD)* bin-packing heuristic, which is likely to create imbalance across processors. Instead, we use the *Worst-Fit Decreasing (WFD)* bin-packing heuristic [28], which selects the least-loaded processor that is a fit for the current task copy. WFD has been shown to reduce imbalance in a related framework [7]; we use it with a similar motive;
- The mapping maps all copies of a task before proceeding to the next task. Instead, we map the first copy of each task, and then the second copy of each task (whenever it exists), and so on. This *layered* approach is expected to: (1) evenly map primary replicas onto processors; (2) decrease the overlap among copies of the same task as long as EDF priority constraints do not call for a full reordering of the tasks during (or after) the mapping. For instance, assume for simplicity that all tasks have the same period, so that the schedule is not constrained by EDF. Then, the mapping of the reference paper will place all replicas of the first task at the beginning of the assignment list of the processors, with possibly more primary replicas on the first processors, while it is better to delay all copies but one and insert first copies of other tasks instead.
- For the dynamic schedule, we keep the idea of running one primary replica of an instance of task τ_i at assigned frequency f_j and to execute all other replicas at frequency f_{max}, but we implement several novel aggressive strategies to reorder and delay chunks.

We outline these modifications step by step below, in Sections 5.3.1 to 5.3.3, which are the respective counterpart, with the same title, of Sections 5.1.2 to 5.1.4. Section 5.3.4 summarizes our new heuristics. Note that we have both online and offline scheduling heuristics. For online scheduling strategies, which are able to determine dynamically the primary copy and secondary copies on the fly, all copies are mapped onto processors using frequency f_j: this is because we must reserve enough room for each copy in the mapping. In other words, we assume that each copy is a primary copy in the mapping, and some of them become secondary only during execution. On the contrary, for offline scheduling strategies, we have decided primary copy and secondary copies during the mapping phase, which means we map one replica at f_j and the rest at f_{max}. In both scenarios, recall that we use different formulas to estimate the energy cost in both cases. Finally, Section 5.3.5 is devoted to complexity results.

5.3.1 Replica sets

For task τ_i at frequency f_j, the reference paper determines the number of copies k_{i,f_j} needed to match the reliability threshold $\mathcal{R}_i$ and estimates the energy cost as $k_{i,f_j} E(\tau_i, f_j, 1)$ where $E(\tau_i, f_j, 1)$ is the energy cost for a single copy. Instead, we propose to estimate the energy cost as:

$$E(\tau_i, f_j, k_{i,f_j}) = E(\tau_i, f_j, 1) + (k_{i,f_j} - 1)E(\tau_i, f_{max}, 1) \tag{5.1}$$

because secondary replicas are actually executed at f_{max} in the reference paper. Equation 5.1 is an accordance with the pessimistic scenario where no replica is cancelled.

Note that since we do not use the same estimation formula for the energy cost as in the reference paper, we may find a different frequency $f_{opt(i)}$ to be used for the mapping step. Again, only one copy (the primary copy) will actually be executed at frequency $f_{opt(i)}$, while all remaining copies (the secondaries) will be executed at frequency f_{max}. For each task τ_i and a given primary frequency f_j, we determine the minimum number of replicas r_i such that the reliability threshold $\mathcal{R}_i$ for each instance is met. Recall that the reliability using a single task instance (no replica) is $R_i(f_j) = \exp^{-\lambda(f_j)c_{i,f_j}}$. If $R_i(f_j) \geq \mathcal{R}_i$, the reliability threshold is enforced, and we need no replica, hence $k_{i,f_j} = 1$. Otherwise, using r additional replicas at f_{max}, the reliability increases to $1-(1-R_i(f_j))(1-R_i(f_{max}))^r$, instead of $1-(1-R_i(f_j))^{r+1}$, and we take the minimal value of r such that this reliability exceeds $\mathcal{R}_i = \sqrt[h_i]{1-\omega(1-R_i(f_{max})^{h_i})}$. This leads to $r = \left\lceil \frac{\log(\frac{1-\mathcal{R}_i}{1-R_i(f_j)})}{\log(1-R_i(f_{max}))} \right\rceil$. The new value of r may be smaller than before, because each replica is more reliable. Of course, the new value of r leads to a new value of $k_{i,f_j} = r+1$. We use this new value of k_{i,f_j} to compute the energy cost. Note that the reference paper assumed that task WCETs are inversely proportional to the assigned frequency where $c_{i,f_j} = \frac{c_i}{f_i}$, but in reality, the interplay between task execution time and frequency is complicated [94]. To be closer to real cases, we let WCETs obey a randomized speedup function similar to Amdhal's law: specifically for task τ_i at frequency f_j, $c_{i,f_j} = s_i c_i + (1-s_i)\frac{c_i}{f_j}$, where the sequential fraction s_i is drawn uniformly and randomly in the interval $[0.1, 0.3]$.

We retain the frequency $f_{opt(i)}$ that minimizes Equation 5.1, and we use $k_{i,f_{opt(i)}}$ copies of task τ_i all running at frequency $f_{opt(i)}$ as input to the mapping phase for all online scheduling strategies. On the contrary, for offline scheduling strategies, we use one copy at frequency $f_{opt(i)}$ and $k_{i,f_{opt(i)}} - 1$ copies at f_{max} as we explained before. Note that we tried through all possible frequencies between f_{min} and $f_{max} = 1$, but the final number of valid frequency levels may be smaller than the number of available frequency levels. There are several possibilities: 1) We should not consider lower frequency levels that makes the task τ_i miss its deadline. 2) As we decrease the frequency level, the number of required replicas may increase or remain the same, the energy consumption is not strictly decreasing. 3) A lower frequency may introduce more overlap that can not be avoided than a higher frequency, so we enforce that $c_{i,f_j} + c_{i,f_{\max}} \leq p_i$.

5.3.2 Mapping and static schedule

We map each replica to a processor while ensuring that no two replicas of the same task are assigned to the same processor. The mapping is done for a whole hyperperiod, with the following constraint for each task: when the first iteration (in case the period differs from the hyperperiod) of a replica is assigned to a given processor, all the other iterations of the same replica will be assigned to the same processor. It allows a simple feasibility check based upon the utilization of all the replicas assigned to the processor.

Recall that any success of a primary replica leads to the immediate cancellation of the secondary replicas, a crucial source of energy saving. The objective of the proposed mapping is thus to avoid overlapping between the execution of the primary and secondary replicas for each task: the primary must be terminated as soon as possible, while the secondaries must be delayed as much as possible. To this end, the mapping strategy ventilates primaries on all processors, in order to minimize conflict among several primary executions. Moreover, it allocates the secondaries while leaving idle time on all processors. This slack can then be used at execution time to delay the execution of the secondaries.

As shown in Algorithm 5, given a list of primary and secondary replicas for each task, and their execution times, we first execute the Worst-Fit Decreasing (WFD) allocation on the primaries ordered

Algorithm 5: Mapping (WFD layer by layer)

Input: The WCET c_i, the period p_i, the assigned frequency f_i, the number of secondary replicas r_i, the number of instances h_i
Output: An allocation of all replicas on the processors σ_m

```
1  begin
2      execute WFD on the primaries considering non-increasing order of c_{i,f_i} × h_i
3      if will do online scheduling afterwards then
4          f_i ← f_i
5      else
6          f_i ← f_max
7      execute WFD on the secondaries layer-by-layer considering non-increasing order of c_{i,f_i} × h_i while ensuring
         that no processor executes more than one replica of each task
8      if ∃m, Σ_{τ_i ∈ σ_m} c_{i,f_i}/p_i > 1 then
9          return not feasible
10     else
11         return σ_m
```

by their total execution time (Line 2). For all secondaries, we assign the same frequency as the primary if preparing for online scheduling afterwards. We assign them the maximum frequency f_{max} otherwise (Line 3-6). Then we execute WFD on the secondaries layer-by-layer and ordered by their total execution time (Line 7), which means we consider the first secondary of all tasks, then the second and so on (as long as it exists). If we successfully map all replicas onto the processors, and the sum of the utilization of each replica on each processor is less than or equal to one (see Section 5.1.3), then we return this allocation. Otherwise, we could not find a feasible mapping with this replica setting (Line 8-11). We observed that the competitor strategy FFD in the reference paper tends to compact all replicas onto the minimum number of processors, while WFD spreads replicas among all available processors, which may give a higher static energy cost. To reduce the influence of static power, we first run WFD with the number of processors used by FFD. If WFD is not able to find a feasible mapping, then we increase the number of processors by one up to the total available number.

We use this allocation mechanism to determine the number of replicas and their frequencies for each task (see Algorithm 6). As described in Section 5.3.1, we already know for each task, at each frequency level, how many replicas are needed to meet the reliability threshold. If we can find a feasible mapping with each task at its energy-optimal frequency $f_{opt(i)}$, then we return this optimal setting (Line 2-7). Otherwise, we check the other end, all tasks run at f_{max} that takes the shortest time possible. If it is still impossible to map all replicas, then there does not exist a feasible mapping (Line 8-13). If there exists a feasible mapping, we will enter the relaxing phase that decreases each primary frequency f_i iteratively until it is no longer possible (Line 14-19). Finally, we return the solution with a frequency level and number of secondary replicas for each task (Line 20).

5.3.3 Dynamic schedule

For the scheduling phase, it is important to start primary replicas as soon as possible, and to delay secondary replicas as much as possible to minimize the overlapping, while still meeting all deadlines. As explained in Section 5.1.4, the reference paper uses the canonical schedule to compute the maximum delay for secondary replicas. Our improvements rely on the following techniques:

- Consider a *scheduling interval* defined by two consecutive deadlines in the global schedule. Inside the interval, task chunks to be executed are ordered by the EDF policy. We observe that we can freely reorder the chunks without missing any deadline, by definition of an interval. It means

Algorithm 6: Replication setting

Input: A set of tasks with cost c_i and reliability requirement $\mathcal{R}_i$
Output: A set of minimum frequency f_i and number of secondary replicas r_i

begin
  /* start with all primaries at energy-optimal frequency */
  for $i \in [1, \ldots, N]$ **do**
    $f_i \leftarrow f_{opt(i)}$
    /* reliability requirement */
    $r_i \leftarrow \left\lceil \frac{\log(\frac{1-\mathcal{R}_i}{1-R_i(f_i)})}{\log(1-R_i(f_{max}))} \right\rceil$
  map the tasks to the processors with Algorithm 5
  if *feasible* **then**
    return $\{f_i, r_i\}$
  /* reset all primaries at $f_{\max}$ */
  for $i \in [1, \ldots, N]$ **do**
    $f_i \leftarrow f_{\max}$
    /* reliability requirement */
    $r_i \leftarrow \left\lceil \frac{\log(\frac{1-\mathcal{R}_i}{1-R_i(f_i)})}{\log(1-R_i(f_{max}))} \right\rceil$
  map the tasks to the processors with Algorithm 5
  if *not feasible* **then**
    return does not exist a feasible mapping
  /* enter the relaxing phase */
  while *any primary frequency can be decreased* **do**
    select task i with LEF or LPF criteria
    $f_i \leftarrow f_{i-1}$
    $r_i \leftarrow \left\lceil \frac{\log(\frac{1-\mathcal{R}_i}{1-R_i(f_i)})}{\log(1-R_i(f_{max}))} \right\rceil$
    map the tasks to the processors with Algorithm 5
    restore f_i and mark task i can not be further decreased if mapping is not feasible
  return $\{f_i, r_i\}$

that in each interval, we should reorder to execute all primary replicas first, and then secondary replicas.

- It is possible to use only a fraction α of each scheduling interval, where α is the utilization. Here is why: at the mapping phase, as long as the total utilization of replicas that are mapped onto the processor is less than or equal to one, then we are able to find a valid scheduling using EDF. Assume we have mapped three tasks t_i, t_j, t_k onto processor p, and that the utilization is $\alpha = \frac{c_{i,f_i}}{p_i} + \frac{c_{j,f_j}}{p_j} + \frac{c_{k,f_k}}{p_k}$. Either we keep the mapping and have a fraction $1 - \alpha$ of the interval where p is idle, or we slow down the execution time of all three tasks by a factor α, then we will have a new utilization $\beta = \frac{c_{i,f_i}}{p_i\alpha} + \frac{c_{j,f_j}}{p_j\alpha} + \frac{c_{k,f_k}}{p_k\alpha} = 1$, which also gives us a feasible mapping without any idle time. This idea can be used in two ways:

  1. Schedule while keeping a fraction $1 - \alpha$ of idleness in each interval. Then, each primary replica is pushed to be beginning of the interval, while secondaries are pushed back to the end of the interval, with idleness in between.
  2. Scale the WCET of all tasks by $\frac{1}{\alpha}$, which also gives a valid canonical schedule, but with longer worst case expected execution time for all tasks. This gives a better reference to further delay the start time of secondary replicas.

- Because we have delayed the start time of secondary replicas, there are some idle slots in the schedule. We take advantage of these idle slots by pre-fetching other primary replica chunks in the availability list: those primaries have been released but were scheduled later because they have lower EDF priority than the current secondary replicas.

5.3.4 Heuristics

Based on the above ideas, we propose several new scheduling heuristics which improve upon EDF_PAPER, the adaptive dynamic scheduling (and the most efficient) heuristic of the reference paper.

EDF_PAPER_PF is an adaptive online scheduling that simply adds the pre-fetching mechanism to EDF_PAPER.

EDF_PAPER_PF_UTILITY is an online scheduling heuristic where we refine EDF_PAPER_PF by using the utilization of each processor. We scale the worst case execution time of all replicas of a given processor by a factor $\frac{1}{\alpha}$, where α is the utilization of that processor.

EDF_IDLE_CEQ is an offline scheduling that builds the EDF schedule for the whole hyperperiod. In each interval defined by two consecutive deadlines, we only use (on each processor) a fraction of the interval defined by the static utilization. It consists in the following steps:

1. consider at each interval the EDF schedule with the constraint of keeping a fraction $1 - \alpha$ of idle time. For secondaries, we refer to the canonical schedule to delay its start time without missing any deadline;

2. start the primary replicas and put aside the secondary replicas in a waiting list to be executed at the end of the interval. Note that for each secondary, as it is impossible to know its actual execution time before its execution, we need to reserve the space for its WCET and to finish execution within the interval;

3. fill in the idle period by inserting other primary replicas that are available;

4. finish the execution of the interval with the secondary replicas in the waiting list, with their actual execution time.

EDF_IDLE_CEQ_ONLINE is the online version of EDF_IDLE_CEQ. It has two major advantages compared to the offline version: (1) we can dynamically decide the primary copy of each task instance, which gives us the flexibility to speed up replicas on the fly; (2) as long as we finish one replica successfully, we can safely cancel other replicas of the same task instance earlier than in static schedules, which gives us more flexibility to adjust the schedule afterwards. Moreover, all algorithms have to reserve for the secondaries some time slots corresponding to their WCET. As their actual execution times are usually shorter, this dynamically frees some time slots that the online schedule uses to prefetch available primary replica chunks.

5.3.5 Complexity analysis

This section is devoted to the proof of several complexity results for the scheduling phase. The global optimization problem is obviously NP-hard, since it is a generalization of the makespan minimization problem with a fixed number of parallel processors [41]. However, the complexity of the sole scheduling phase is open: if the number or replicas has already been decided for each task, and if the frequency and assigned processor of each replica has also been decided, the sole scheduling phase aims at minimizing the expected energy consumption. In this section, we first state a lower bound for this scheduling problem, and then we assess the complexity of achieving this lower bound. We show that the instance with identical WCETs is polynomial, while the instance with different WCETs is NP-complete in the strong sense.

Lower bound

Consider the following instance of the scheduling phase:

- All tasks have the same period p, hence there is a single instance of each task in the hyperperiod of length $L = p$. hence EDF constraints do not apply, and each task is scheduled without preemption (as a single chunk)
- $F = 1$: there is a unique frequency $f_{max} = 1$
- For $1 \leq i \leq n$, task τ_i has k_i replicas, including itself. The j-th replica, with $1 \leq j \leq k_i$, is mapped onto processor $P_{alloc(i,j)}$, where$1 \leq alloc(i,j) \leq M$. For each task τ_i, replicas are mapped onto different processors: $alloc(i,j_1) \neq alloc(i,j_2)$ for $1 \leq j_1 < j_2 \leq k_i$.
- The WCET of any replica of task τ_i is c_i, its reliability is $R_i = R_i(f_{max})$, and its consumed energy is $E_i = E(\tau_i, f_{max}, 1)$.
- $P_{static} = 0$, meaning that no energy is spent when a processor is idle

Thus, each of the M processors has a list of assigned replicas to execute. It can choose any ordering because all tasks have the same period, hence all deadlines will be enforced, regardless of the ordering. We further assume that the mapping is valid, which translates on each processor P_q, $1 \leq q \leq M$, by the condition:

$$\sum_{1 \leq i \leq n, 1 \leq j \leq k_i, alloc(i,j)=q} WCET_i \leq p$$

Proposition 5.1. *A lower bound on the total expected energy consumed is*

$$E_{LB} = \sum_{i=1}^{n} \left(E_i \sum_{j=1}^{k_i} (1 - R_i)^{j-1} \right) \tag{5.2}$$

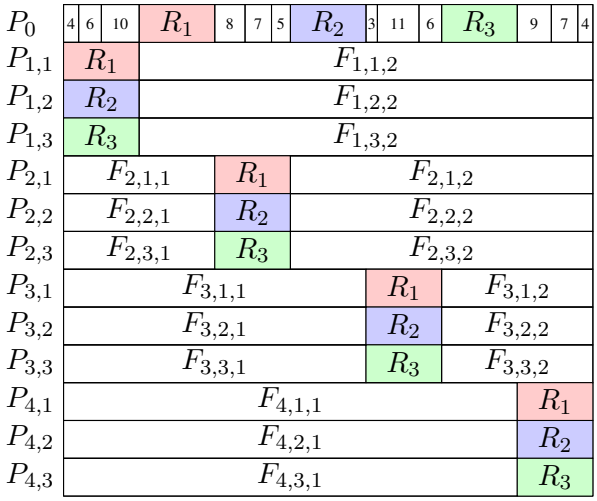


Figure 5.3: Scheduling for a solution of $\mathcal{I}$ of 3-Partition, with $m = 4$, $B = 20$, and $(a_1, ..., a_{12}) = (4, 3, 8, 9, 7, 6, 11, 6, 7, 10, 4, 5)$. On processor P_0 the digits are the sizes of the A_i's.

The bound E_{LB} is met if and only if the scheduling achieves no overlap between any two replicas of the same task.

Proof. For each task, we need to execute the replica which is scheduled in first position. If this replica fails, with probability $1 - R_i$, we need to execute the replica which is scheduled in second position. If both replicas fail, we need to execute the replica which is scheduled in third position, and so on. This directly leads to the lower bound E_{LB}.

Now if any two replicas of the same task, say τ_i, do overlap, then with some non-zero probability, both replicas will execute, and the consumed energy will be strictly higher than the contribution of τ_i to Equation (5.2). This concludes the proof. □

Identical costs

Proposition 5.2. *When all tasks have the same WCET ($c_i = c$ for $1 \leq i \leq n$), one can build a schedule meeting the lower bound of Equation (5.2) in polynomial time.*

Proof. We construct a bipartite graph with tasks on the left (with N vertices) and processors on the right (with M vertices). Task τ_i has k_i edges, connecting each of its replicas to the assigned processor. Hence we have $K = \sum_{i=1}^{n} k_i$ edges. Let δ be the maximum degree of a vertex in the graph. According to Konig's edge coloring theorem, one can find a collection of δ perfect matchings that cover all edges in the graph, in time $O(\delta K)$ [87]. Since $\delta K \leq (n + M)^3$, this is indeed polynomisal in the problem size. These mappings directly lead to a schedule with minimal makespan δc. By construction, this schedule is guaranteed overlap-free. □

Arbitrary costs

Proposition 5.3. *When tasks have different WCET, determining whether the lower bound of Equation (5.2) can be met, is a problem which is NP-complete in the strong sense.*

Proof. Let NOOVERLAP denote the problem with different WCETs. NOOVERLAP clearly belongs to the class NP: a certificate can be the description of the schedule with start and end times for each replica, and one can check in quadratic time that no two replicas of the same task overlap. We establish completeness in the strong sense through a reduction from 3-Partition [41]. Let $\mathcal{I}$ be an instance of 3-Partition. $\mathcal{I}$ comprises $3m$ integers, $a_1, ..., a_{3m}$ such that $\sum_{i=1}^{3m} a_i = mB$ and $\frac{B}{4} < a_i < \frac{B}{2}$ for all i. The question is: can we partition the a_i's into m subsets $S_1, ..., S_m$ such that each subset has total size B: $\sum_{j \in S_i} a_j = B$? The size of $\mathcal{I}$ is $O(m+B)$. From the instance $\mathcal{I}$ of 3-Partition, we build an instance $\mathcal{J}$ of NOOVERLAP: this instance contains three types of tasks: some replicated tasks (the R_i's), some filling tasks which constrain the replicated tasks (the $F_{i,j}$'s), and the tasks corresponding to the integers in instance $\mathcal{I}$ (the A_i's). Specifically:

- There are $1 + m(m-1)$ processors denoted P_0 and $P_{i,j}$ with $1 \leq i \leq m$ and $1 \leq j \leq m-1$.
- All tasks have the same period $p = (2m-1)B$.
- There are $m-1$ replicated tasks of size B, $R_1, ..., R_{m-1}$. For any i, $1 \leq i \leq m-1$, there is one replica of R_i on processor P_0, and 1 on each of the processors $P_{k,i}$, $1 \leq k \leq m$. Thus, each R_i's is replicated $m+1$ times.
- Two tasks $F_{i,j,1}$ and $F_{i,j,2}$ are mapped on each processor $P_{i,j}$, $1 \leq i \leq m-1$, $1 \leq j \leq m$. $F_{i,j,1}$ is of size $2(i-1)B$ and $F_{i,j,2}$ is of size $2(m-i)B$. The total load of processor $P_{i,j}$ is $B + 2(i-1)B + 2(m-i)B = (2m-1)B$ and there is no slack on that processor. Note that, to ease the writing, we have kept in our description a null size task on processors $P_{i,1}$ and $P_{i,m}$.
- In addition to one replica of each of the tasks R_1, R_{m-1}, $3m$ tasks $A_1, ..., A_{3m}$ are mapped to processor P_0, where task A_i has size a_i. Therefore, the total load of processor P_0 is $(m-1)B + \sum_{i=1}^{3m} a_i = (2m-1)B$ and there is no slack on that processor either.

Instance $\mathcal{J}$ contains $1+m(m-1)$ processors and $(m-1)(m+1)+2m(m-1)+3m = 3m^2+3m-1$ replicas. All tasks have size $O(mB)$. Hence, the size of $\mathcal{J}$ is polynomial in the size of $\mathcal{I}$. We now prove that if $\mathcal{I}$ has a solution, then $\mathcal{J}$ has a solution. Let $S_1, ..., S_m$ be the solution of $\mathcal{I}$. Then we schedule the tasks of $\mathcal{J}$ as follows and as illustrated by Figure 5.3:

- For any i, $1 \leq i \leq m-1$, and any j, $1 \leq j \leq m$, on processor $P_{i,j}$, task $F_{i,j,1}$ is executed during the interval $[0, 2(i-1)B]$, a replica of R_j during the interval $[2(i-1)B, (2i-1)B]$, and $F_{i,j,2}$ during the interval $[(2i-1)B, (2m-1)B]$.
- For any i, $1 \leq i \leq m-1$, a replica of R_i is executed on P_0 during the interval $[(2i-1)B, 2iB]$.
- For any j, $1 \leq j \leq m$, the tasks corresponding to the j-th partition of $\mathcal{I}$, i.e., the tasks A_k such that $k \in S_j$, are executed on P_0 during the time interval $[2(j-1)B, (2j-1)B]$.

One can easily check that this schedule is valid and that two replicas of a task R_i are never executed simultaneously. Therefore there exists a schedule without overlap for $\mathcal{J}$ if there exists a solution for $\mathcal{I}$.

Let us now assume that there exists a valid schedule for $\mathcal{J}$, i.e., a schedule without any overlap. Let us consider any i, $1 \leq i \leq m-1$. Processor $P_{i,1}$ (respectively $P_{i,m}$) contains a replica R_i and a task of size $(2m-2)B$ (we do not care about the null-size task). Therefore, the replica R_i is executed on $P_{i,1}$ (resp. $P_{i,m}$) either during the interval $[0, B]$ or during $[(2m-2)N, (2m-1)B]$, Then, for any j, $1 < j \leq \frac{m}{2}$, processor $P_{i,j}$ (respectively $P_{i,m-j+1}$) contains a replica R_i, a task of size $(2i-1)B$ and one of task $2(m-i)B$. From what precedes, there is already a replica of R_i executed during the time interval $[0, B]$ and one during $[(2m-2)B, (2m-1)B]$. Therefore, the replica R_i is executed on $P_{i,j}$

(resp. $P_{i,m-j+1}$) either during the interval $[(2i-1)B, 2iB]$ or during $[(2m-i)B, (2m-i+1)B]$. Overall, on processor P_0, the replica R_i must be executed during one of the intervals $[(2j-1)B, (2j)B]$, for $1 \leq j \leq m-1$, because at all the other instants, there is already one R_i replica being executed on one other processor, and because the schedule is without any overlap. However, there are $m-1$ such intervals and $m-1$ such replicas. Therefore, the tasks A_i's must be executed during the intervals $[(2j-2)B, (2j-1)B]$, for $1 \leq j \leq m$. This is a set of m intervals each of size B. Because the schedule is valid, all the A_i's are executed during these intervals. Let S_j, $1 \leq j \leq m$, be the set of the indices of the A_l's executed during $[(2j-2)B, (2j-1)B]$. Then the subsets S_j define a solution to $\mathcal{I}$. □

5.4 Performance evaluation

In this section, we present the simulation results to evaluate the performance of our whole strategy compared to the best solution proposed in the reference paper [54]. In Section 5.4.1, we describe the parameters and settings used during the experimental campaign. We present the results in Section 5.4.2.

5.4.1 Experimental methodology

We designed a discrete event simulator, which is publicly available at [51]. For each data point, we considered $2,000$ data sets with 20 tasks. Task periods are randomly generated between 10ms and 100ms. The utilization of each task is generated randomly using the UUnifast scheme [14], with the total utilization U_{tot} as input. The set of frequencies is taken from a real microprocessors [85] which has $F = 5$ frequency levels including 1.0, 0.8, 0.6, 0.4 and 0.15. We also validate our methods on two other real frequencies sets [40, 94]. Following [54], in the rest of the section, we assume that the transient fault arrival rate at f_{max} is $\lambda_0 = 10^{-6}$ and the system sensitivity factor $d = 4$. The static power and the frequency-independent are set to 5% and 15% respectively of the maximum frequency-dependent power consumption with $C = 1$. We use the ratio $\beta_{\text{b/w}}$ of the best-case over the worst-case execution time, to model workload variability. The actual execution time of each task instance is determined according to a uniform distribution between $\beta_{\text{b/w}} c_i$ and c_i. To compare the strategies under all parameter settings, we covered all of the variables from [54]. For the rest of the section, we keep the number of cores at 8, vary the value of U_{tot} from 1.5 to 3.5 to study the impact of system load, vary $\beta_{\text{b/w}}$ from 0.2 to 1.0 to show the influence of workload variability, and vary the probability of failure scaling factor w from 10^{-5} to 10^{-1} to evaluate the impact of target reliability. We consider heuristic LPF for choosing the candidate task for relaxation, which is shown to be the best heuristic in [54]. However, a full set of results supplemented with two additional real frequencies sets and a larger range s_i, $s_i \in [0.1, 0.5]$ can be found in [84]. All results show the same trends as the results described below.

5.4.2 Results

We use the same *baseline* scheme as the reference paper[2], i.e., classic EDF scheduling with First-Fit Decreasing mapping and replica sets from [54] (where it is called the *static scheme*). Figures 5.4 and 5.5 present the energy consumption of our strategies and of the best performance strategy from the reference paper, divided by the energy consumption of the baseline. Therefore, the lower the better and data points below the $y = 1$ line denote cases in which these strategies outperform the static scheme (i.e., achieve a lower energy consumption). These figures represent the trends with various parameters.

[2]The authors of [54] have not provided their source code to us; we did our best to ensure a fair assessment and comparison of results.

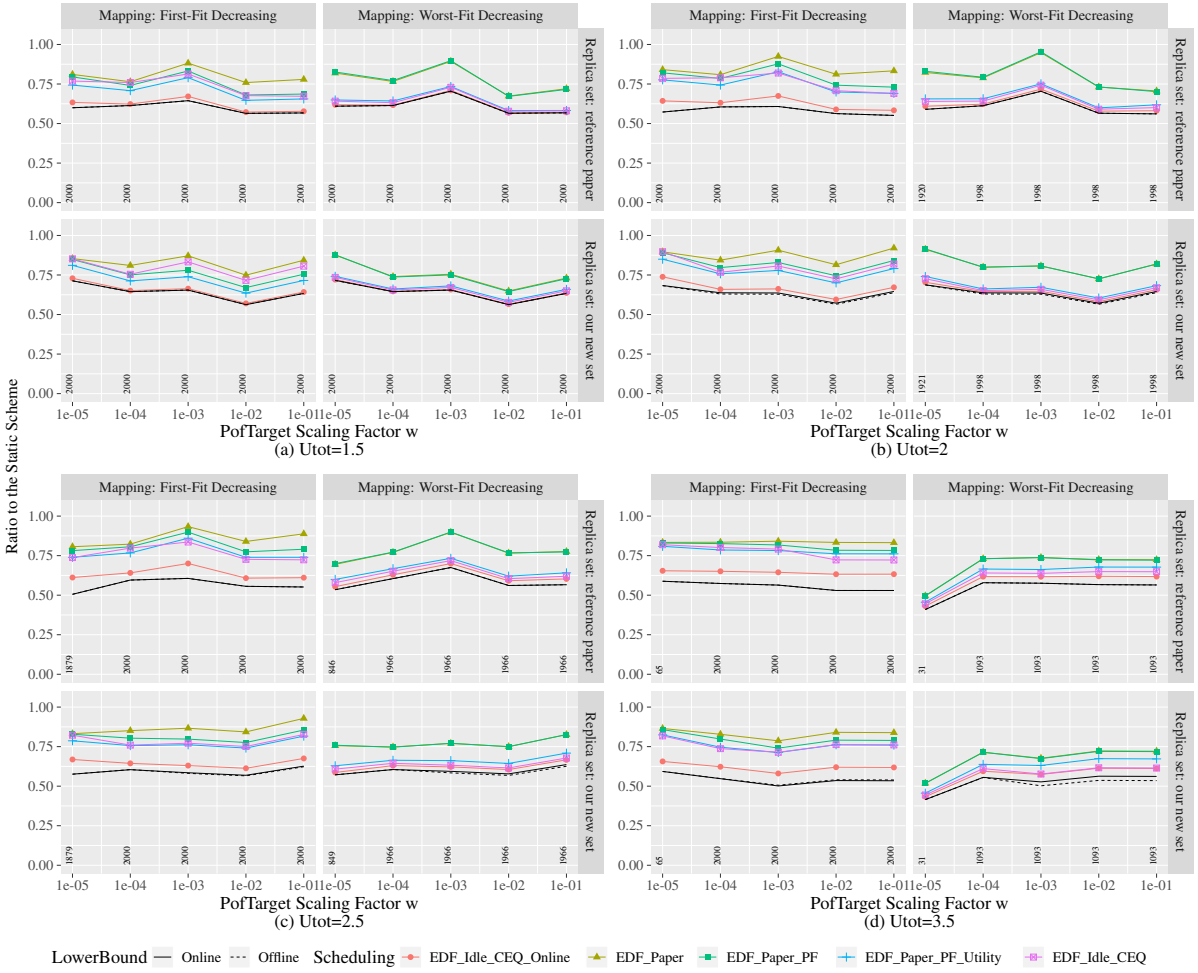


Figure 5.4: Impact of U_{tot} and w with $\beta_{\text{b/w}} = 1$.

Each subfigure shows results for a different combination of mapping and replica settings. Each line of subfigures is for a different replica sets (either from the reference paper or our new set) while each column is for a different mapping (FFD and WFD). For example, the bottom right plot presents energy savings of several scheduling approaches (different line colors) with WFD mapping and our replica setting. We report in these figures the number of seeds (out of 2000 in total) that could find a feasible solution for each setting. These numbers are reported in black above the horizontal axis in each figure. Note that WFD tends to find fewer or the same number of feasible solutions than FFD. This is because FFD tries to pack more tasks onto processors while WFD tends to spread tasks onto less loaded processors. Moreover, we plot the lower bound for online scheduling and offline scheduling (different line styles) in black lines, by which we could know the maximum energy saving that can be achieved without any overlapping and failure. We can notice that when we apply our replica setting, the lower bound of the offline scheduling is sometimes slightly lower than that of the online scheduling. The reason is that during the mapping phase, in the online scheduling, we assign all secondaries the same frequency as the primary, which makes it more difficult to find a feasible mapping than in the offline scheduling that assigns secondaries the maximum frequency. Hence, the offline scheduling may pick a lower frequency (see Section 5.3.2).

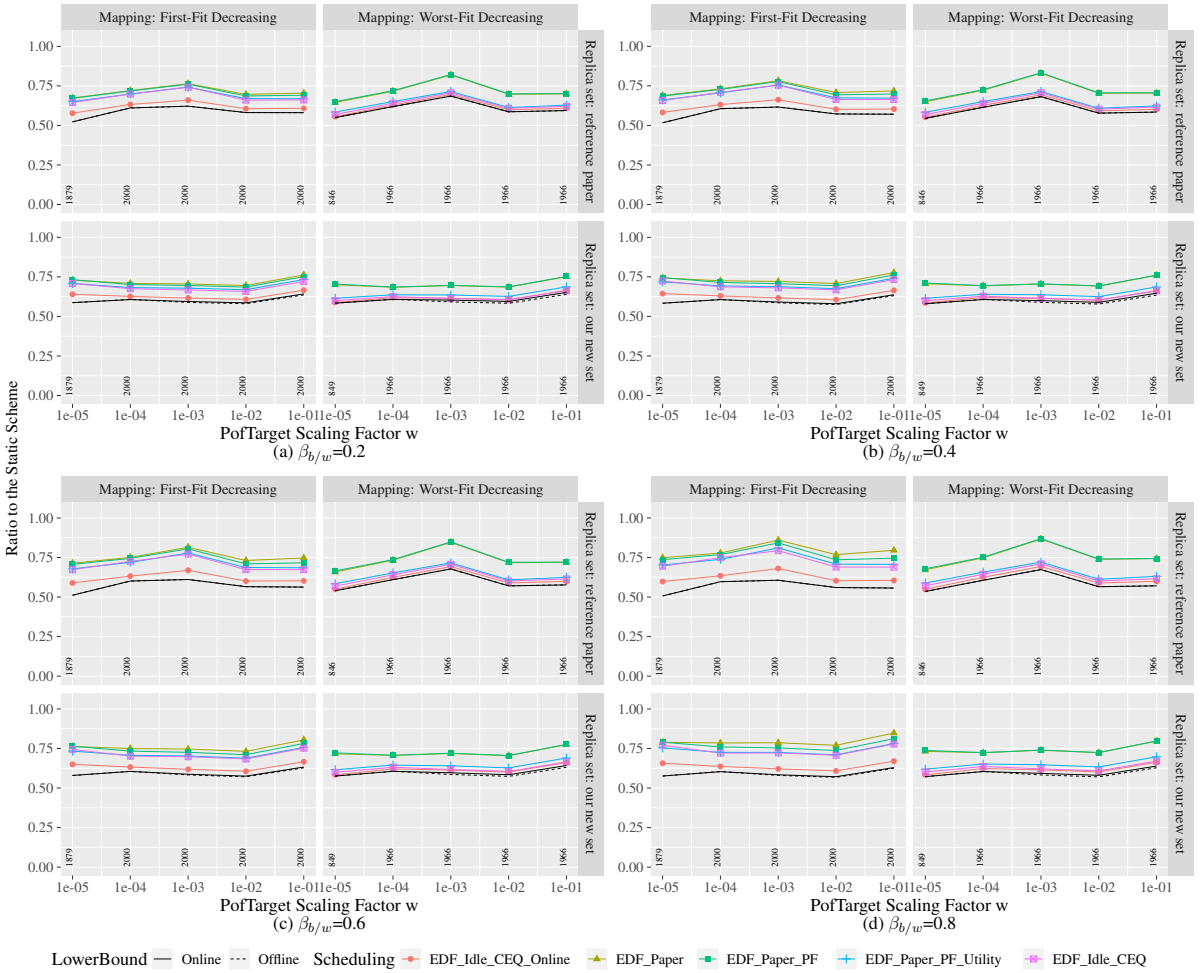


Figure 5.5: Impact of $\beta_{\text{b/w}}$ and w with $U_{tot} = 2.5$.

A clear observation is that our scheduling heuristics outperform EDF_PAPER under almost all combinations of settings. In particular, EDF_IDLE_CEQ_ONLINE always achieves the best performance. EDF_IDLE_CEQ_ONLINE could save up to 25% with respect to EDF_PAPER (see Figure 5.4(c)). It should be noted that to clearly see the different capabilities of the scheduling heuristics in reducing the overlapping, we need more than one replica for each task. As we decrease the value of w, more replicas are needed to meet the reliability target. In Figure 5.4, we keep the value of $\beta_{\text{b/w}}$ at 1 and increase the system load. We observe that the energy savings are closer to the lower bound at lower utilization, as we can find a feasible partitioning without overlapping for the minimum energy configurations. In Figure 5.5, for a fixed system load ($U_{tot} = 2.5$), the higher the $\beta_{\text{b/w}}$ ratio, the lower the workload variability, and the further from the lower bound. This is because as $\beta_{\text{b/w}}$ increases, jobs have larger execution times and there is a higher chance of overlap between the primary and the secondary replicas.

Another common trend is that, in the majority of cases, WFD and our new replica setting help save energy. By considering our replica setting, we could find a lower $f_{opt(i)}$. This is because our replica setting plans for one replica at f_i and x replicas at f_{max} while the reference setting plans to run all replicas at f_i. By considering WFD, we usually: 1) achieve the highest energy savings, 2) improve the average performance of scheduling heuristics. For example, in Figure 5.4(a), from the two plots in the second row (with the same replica set but with different mappings), we can see that more scheduling heuristics achieve the lower bound with WFD than with FFD (with whom only EDF_IDLE_CEQ_ONLINE did). In Figure 5.4(d), we could see that applying the replica sets and mapping of the reference paper (top left plot), the best energy savings are achieved by EDF_IDLE_CEQ_ONLINE (around 37.5%), while the other heuristics save less that 25%. With our replica sets and mapping (bottom right), our scheduling heuristics achieve more than 40% energy savings. Even the EDF_PAPER saves up to 25%. Thus, our solutions outperform those of the reference paper. These improvements are representative of the trends that can be observed for all considered graphs, but suffer from some exceptions. For example, in Figure 5.4(a), when $w = 10^{-5}$, applying our replica set gives a higher lower bound which means we will lose even without any overlapping in the schedule. Overall, our task mapping and replica setting never achieves significantly bad performance, and most of the time achieves the best performance. The average improvement of our method over [54], computed across all our experiments, can be estimated to be of the order of 20%.

5.5 Conclusion

In this chapter, we have revisited the challenging problem presented by Haque, Aydin and Zhu in [54], namely minimizing the expected energy consumption of a set of preemptive periodic real-time tasks, executing on a parallel platform where processors are subject to transient failures. Replication is used to enforce the reliability threshold, and all deadlines should be met as well. We have improved the approach of [54] as follows. First, we use a different formula to estimate the energy consumption, which is supposed to be closer to actual execution scenarios. Secondly, in the mapping phase, we apply a layered WFD strategy, which is expected to be helpful for load-balancing, and for decreasing the overlap among copies of the same task. Finally, we implement several novel scheduling strategies, which introduce the idea of reordering chunks in between deadlines and of taking advantage of the utilization of the processor. Moreover, we have established that the sole problem of scheduling tasks with different WCETs, knowing the number of replicas, frequency and assigned processor for each task, is NP-complete in the strong sense. We have evaluated the improvement of our strategies with a discrete event simulator (made publicly available). Extensive experiments conducted for various range of parameters have shown that: 1) Our new replica sets and WFD help finding a lower frequency,

which gives good pre-condition for further energy savings; 2) our scheduling heuristics significantly outperform EDF_PAPER. More specifically, EDF_IDLE_CEQ and EDF_IDLE_CEQ_ONLINE have the best performance under a wide range of scenarios, with an average gain in energy of 20%.

In the next chapter, we will deal with the same problem with independent tasks, but targeting heterogeneous multicore systems.

Chapter 6

Heterogeneous platforms

As stated in Section 4.1, the design of the mapping and scheduling heuristics to solve the tri-criteria problem on heterogeneous platforms is much more technical than in the case of identical processors studied in Chapter 5. This is because the reliability of a replica of task τ_i depends upon the processor which executes it. More precisely, the reliability of a replica of task τ_i mapped on processor m_k is $R(\tau_i, m_k) = e^{-\lambda_k c_{i,k}}$, where $c_{i,k}$ is the execution time of τ_i on m_k, and λ_k the failure rate of m_k. The total reliability of τ_i is a function of the reliability of all replicas (which we will explicit in Equation 6.4); hence, it is not known until the end of the mapping process (unless we pre-compute an exponential number of reliability values). Then there are many processors to choose from, and those providing a high reliability, thereby minimizing the number of replicas needed to match the reliability threshold, may also require a high energy cost per replica: in the end, it might be better to use less reliable but also less energy-intensive processors. Furthermore, the reliability is not enough to decide for the mapping: if two processors offer similar reliabilities for a task, it might be better to select the one with smaller execution time, in order to increase the possibility of mapping other tasks without exceeding any deadline. Altogether, we face a more complicated decision than Chapter 5, and we provide several criteria to guide the mapping process.

To assess the performance of our heuristics, we use a comprehensive set of execution scenarios, with a wide range of processor speed profiles and failure rates. When the failure rate is low, most heuristics are equivalent, but when the failure rate is higher, only a few heuristics achieve a good performance. Because we have no guarantee on the performance of the global mapping and scheduling process, we analytically derive a lower bound for the expected energy consumption of any mapping. This bound assumes no overlap between any two replicas of the same task, and it cannot always be met because of utilization constrains. Nevertheless, we show that the performance of our best heuristics remain quite close to this bound in the vast majority of simulation scenarios. The main contributions of this chapter are the following:

- The formulation of the tri-criteria optimization problem on heterogeneous platforms;
- The design of several mapping and scheduling heuristics;
- The characterization of a lower bound for performance;
- An experimental evaluation based on a comprehensive set of simulations scenarios, showing that two of the heuristics always achieve the best performance, and are always very close to the lower bound.

The rest of the chapter is organized as follows. Section 6.1 provides a detailed description of the optimization problem under study, including a few notes on its complexity. The mapping and scheduling heuristics are described in Section 6.2 and Section 6.3 respectively. The performance lower bound is

introduced in Section 6.4. Section 6.5 is devoted to a comprehensive experimental comparison of the heuristics. Finally, Section 6.6 gives concluding remarks and hints for future work.

Table I: Key Notations

Notation	Explanation
N and M	number of tasks and of processors
p	period (deadline) for each task instance
$c_{i,k}$	WCET for task τ_i on processor m_k
$u_{i,k} = \frac{c_{i,k}}{p}$	utilization of task τ_i executing on processor m_k
u_k	utilization of m_k (sum of utilization of replicas assigned to m_k)
$\mathcal{R}_i$	target reliability threshold for task τ_i
λ_k	failure rate of processor m_k
$P(m_k)$	power consumed per time unit on processor m_k
E_s	total static energy consumption
$E_d(\tau_i, m_k)$	dynamic energy cost of task τ_i on processor m_k
$R(\tau_i, m_k)$	reliability of task τ_i on processor m_k

6.1 Model

The inputs to the optimization problem are a set of real-time independent tasks, a set of non-identical processors and a reliability target. Key notations are summarized in Table I.

6.1.1 Platform and tasks

The platform consists of M heterogeneous processors $m_1, m_2, \ldots, m_M$ and a set of N periodic atomic tasks $\tau_1, \tau_2, \ldots, \tau_N$. Each task τ_i has WCET $c_{i,k}$ on the processor m_k. The WCETs among different processors are not necessarily related. In the experiments, we generate the $c_{i,k}$ values with the method proposed in [20], where we have two parameters to control the correlation among task execution times and processors (see Section 6.5.1 for details). Each periodic task τ_i generates a sequence of *instances* with period p, which is equal to its deadline. In this work, we assume that all tasks have the same period p, so that a single instance of each task must execute every p seconds. Note that assuming that all tasks are atomic and with same period is the standard assumption for real-time task graphs (or DAGs) [113].

As already mentioned, real-time tasks usually complete execution earlier than their estimated WCET: execution times are assumed to be data-dependent and non-deterministic, randomly sampled from some probability distribution whose support is upper bounded by the WCET. See Section 6.5.1 for details on the generation of actual execution times from WCET values. The *utilization* $u_{i,k}$ of task τ_i executing on processor m_k is defined as $u_{i,k} = \frac{c_{i,k}}{p}$. The utilization of a processor is the sum of the utilizations of all tasks that are assigned to it.

6.1.2 Power and energy

The power consumed per time unit on processor m_k is expressed as

$$P(m_k) = P_{k,s} + gP_{k,d} \tag{6.1}$$

where $P_{k,s}$ is the static power; g represents the system state and indicates whether dynamic power $P_{k,d}$ is currently being consumed by m_k: when m_k executes a task, $g = 1$, otherwise $g = 0$. To summarize,

we have $2M$ input values, $\{P_{1,s}, P_{2,s} \dots P_{M,s}\}$ for static powers and $\{P_{1,d}, P_{2,d} \dots P_{M,d}\}$ for dynamic powers.

The dynamic energy consumption $E_d(\tau_i, m_k)$ of task τ_i on processor m_k is estimated using the WCET:

$$E_d(\tau_i, m_k) = P_{k,d} \times c_{i,k} \tag{6.2}$$

but we use the value derived from the actual execution time in the experiments. The total static energy consumption is simply given by

$$E_s = \sum_{k \in Used} P_{k,s} \times p \tag{6.3}$$

where *Used* denotes the index set of the processors used by the schedule.

6.1.3 Reliability

We consider transient faults, modeled by an Exponential probability distribution of rate λ_k on processor m_k. Thus, fault rates differ from one processor to another. This is a very natural assumption for a heterogeneous platform made of different-type processors. At the end of the execution of each task, there is an *acceptance test* to check the occurrence of soft errors induced by the transient faults. It is assumed that acceptance tests are 100% accurate, and that the duration of the test is included within the task WCET [54].

The *reliability* of a task instance is the probability of executing it successfully, in the absence of permanent faults. The reliability of task τ_i on processor m_k with WCET $c_{i,k}$ is $R(\tau_i, m_k) = e^{-\lambda_k \times c_{i,k}}$. During the mapping phase, task τ_i will have several replicas executing on different processors, in order to match some reliability threshold. Let $alloc(i)$ denote the index set of the processors executing a replica of τ_i. The mapping achieves the following reliability $R(\tau_i)$ for task τ_i:

$$R(\tau_i) = 1 - \Pi_{k \in alloc(i)}(1 - R(\tau_i, m_k)) \tag{6.4}$$

Indeed, the task will succeed if at least one of its replicas does: the success probability is thus equal to 1 minus the probability of all replicas failing, which is the expression given in Equation 6.4.

Each task τ_i has a reliability threshold $\mathcal{R}_i$ which is an input of the problem and that must be met by the mapping. In other words, the constraint writes $R(\tau_i) \geq \mathcal{R}_i$ for $1 \leq i \leq N$. Because the tasks are independent, it is natural to assume that they might have different reliability thresholds: a higher threshold means that more resources should be assigned for the task to complete successfully with a higher probability. In the experiments we use $\mathcal{R}_i = \mathcal{R}$ for all tasks, but our heuristics are designed to accommodate different thresholds per task.

6.1.4 Optimization objective

The objective is to determine a set of replicas for each task, a set of processors to execute them, and to build a schedule of length at most p, so that expected energy consumption is minimized, while matching the deadline p and reliability threshold $\mathcal{R}_i$ for each task τ_i.

As already mentioned in the introduction, the expected energy consumption is an average made over all possible execution times randomly drawn from their distributions, and over all failure scenarios (with every component weighted by its probability to occur). An analytical formula is out of reach, and we use Monte-Carlo sampling in the experiments. However, we stress the following two points:

- To guide the design of the heuristics, we use a simplified objective function; more precisely, we use WCETs instead of (yet unknown) actual execution times, and we conservatively estimate the

dynamic energy of a task as the sum of the dynamic energy of all its replicas. Because mapping decisions are based upon WCETs, the number of enrolled processors does not depend upon actual execution times and the static energy is always the same for all scenarios, namely the length of the period times the sum of the static powers of the enrolled processors (see Equation 6.3).

- To assess the absolute performance of the heuristics, we derive a lower bound for the energy consumption. This bound is based upon actual execution times but neglects scheduling constraints and assumes no overlap between any two task replicas, hence it is not reachable in general. However, we show that our best heuristics achieve performance close to this bound.

6.1.5 Complexity

The global optimization problem is obviously NP-hard, since it is a generalization of the makespan minimization problem with a fixed number of parallel processors [41]. The optimization of the sole scheduling phase is also NP-hard: if the number of replicas has already been decided for each task, and if the assigned processor of each replica has also been decided, the scheduling phase aims at minimizing the expected energy consumption by avoiding overlap between the replicas of a same task (proved in Section 5.3.5). Even if the task deadline was not constraining (very large deadline with respect to the worst-case execution time of tasks), the problem would remain NP-hard. We formally state this latter problem and show that it is NP-hard.

Definition 6.1 (MINENERGYMAXRELIABILITY)**.** *Consider an heterogeneous platform composed of M heterogeneous processors, $m_1, ..., m_M$, and N (non-periodic) tasks $\tau_1, \tau_2, ..., \tau_N$. Executing task τ_i on processor m_k has an energy cost of $E(\tau_i, m_k)$ and has a probability of success of $R(\tau_i, m_k)$. Let $\mathcal{E}$ and $\mathcal{R}$ be two constants. The* MINENERGYMAXRELIABILITY *decision problem is: is it possible to schedule the tasks on the processors so that: (i) the total energy consumed does not exceed $\mathcal{E}$; and (ii) the probability that all tasks succeed is at least $\mathcal{R}$?*

Lemma 6.1. *Problem* MINENERGYMAXRELIABILITY *is NP-complete.*

Proof. We prove this result by a reduction from the 2-PARTITION problem [41]. Let I_1 be an instance of 2-PARTITION with N positive integers, $a_1, ..., a_N$. Let $S = \sum_{i=1}^{N} a_i$. The question is: is it possible to find a subset A of $\{1, ..., N\}$ such that

$$\sum_{\substack{1 \leq i \leq N \\ i \in A}} a_i = \sum_{\substack{1 \leq i \leq N \\ i \notin A}} a_i = \frac{S}{2}.$$

From $\mathcal{I}_1$ we build an instance $\mathcal{I}_2$ of MINENERGYMAXRELIABILITY as follows. We have $M = 2$ processors. Then we have N tasks each having the same execution times on both processors: $c_{i,1} = c_{i,2} = a_i$. The failure rates are defined by: $\lambda_1 = \frac{1}{S}$ and $\lambda_2 = 1$. The static energy is null, $P_{1,s} = P_{2,s} = 0$, and the dynamic energy is defined by $P_{1,d} = 1$, $P_{2,d} = \frac{1}{S}$. Therefore, we have:

$$\begin{cases} E(\tau_i, m_1) = a_i \text{ and } R(\tau_i, m_1) = e^{-\frac{a_i}{S}} \\ E(\tau_i, m_2) = \frac{a_i}{S} \text{ and } R(\tau_i, m_2) = e^{-a_i} \end{cases}$$

Finally, we let $\mathcal{E} = \frac{1}{2}(S + 1)$ and $\mathcal{R} = e^{-\frac{1}{2}(S+1)} = e^{-\mathcal{E}}$.

One can easily check that the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$, that all the $E(\tau_i, m_k)$'s are positive and that all the $R(\tau_i, m_k)$'s are strictly between 0 and 1.

Let us consider any mapping of the M tasks on the two processors. Let A be the index set of tasks mapped on processor 1 in this mapping, and let $\mathcal{A} = \sum_{i \in A} a_i$. The total energy E consumed by this mapping is:

$$\begin{aligned} E &= \left(\sum_{\substack{1 \leq i \leq N \\ i \in A}} E(\tau_i, m_1) \right) + \left(\sum_{\substack{1 \leq i \leq N \\ i \notin A}} E(\tau_i, m_2) \right) \\ &= \left(\sum_{\substack{1 \leq i \leq N \\ i \in A}} a_i \right) + \left(\sum_{\substack{1 \leq i \leq N \\ i \notin A}} \frac{a_i}{S} \right) \\ &= \mathcal{A} + \frac{1}{S}(S - \mathcal{A}) \\ &= \mathcal{A}\left(1 - \frac{1}{S}\right) + 1 \\ &= \mathcal{A}\frac{S-1}{S} + 1 \end{aligned}$$

The reliability R of the whole set of tasks is:

$$R = \left(\prod_{\substack{1 \leq i \leq N \\ i \in A}} R(\tau_i, m_1) \right) \times \left(\prod_{\substack{1 \leq i \leq N \\ i \notin A}} R(\tau_i, m_2) \right)$$

Taking logarithms, we obtain:

$$\begin{aligned} \ln(R) &= \left(\sum_{\substack{1 \leq i \leq N \\ i \in A}} ln(R(\tau_i, m_1)) \right) + \left(\sum_{\substack{1 \leq i \leq N \\ i \notin A}} ln(R(\tau_i, m_2)) \right) \\ &= \left(\sum_{\substack{1 \leq i \leq N \\ i \in A}} \frac{-a_i}{S} \right) + \left(\sum_{\substack{1 \leq i \leq N \\ i \notin A}} (-a_i) \right) \\ &= \left(-\frac{\mathcal{A}}{S} \right) + (-(S - \mathcal{A})) \\ &= -S + \mathcal{A}(1 - \frac{1}{S}) \\ &= -S + \mathcal{A}\frac{S-1}{S} \end{aligned}$$

Now let us assume that $\mathcal{I}_2$ has a solution:

$$\begin{cases} E \leq \mathcal{E} \\ R \geq \mathcal{R} \end{cases} \Leftrightarrow$$

$$\begin{cases} E \leq \mathcal{E} \\ \ln(R) \geq \ln(\mathcal{R}) \end{cases} \Leftrightarrow$$

$$\begin{cases} \mathcal{A}\frac{S-1}{S} + 1 \leq \frac{1}{2}(S+1) \\ -S + \mathcal{A}\frac{S-1}{S} \geq -\frac{1}{2}(S+1) \end{cases} \Leftrightarrow$$

$$\begin{cases} \mathcal{A}\frac{S-1}{S} \leq \frac{1}{2}(S-1) \\ \mathcal{A}\frac{S-1}{S} \geq \frac{1}{2}(S-1) \end{cases} \Leftrightarrow$$

$$\begin{cases} \mathcal{A} \leq \frac{S}{2} \\ \mathcal{A} \geq \frac{S}{2} \end{cases}$$

Therefore, $\mathcal{I}_2$ has a solution if and only if $\mathcal{I}_1$ has a solution. □

6.2 Mapping

In the mapping phase, we need to define the number of replicas for each task, as well as the execution processor for every replica, aiming at meeting the reliability target while minimizing the energy cost. One difficulty introduced by platform heterogeneity is that we do not know the number of replicas needed for each task to reach its reliability threshold, before completing the mapping process, because different processors have different failure rates and speeds and, hence, they provide different reliabilities for each replica. Therefore, the simpler three-step method of Chapter 5 cannot be applied.

As shown in Algorithm 7, given a set of tasks with their reliability targets and a set of heterogeneous processors, we first order the tasks according to TASKMAPCRITERIA, which includes:

- *deW* (*inW*): decreasing (increasing) average work size $\bar{c_i} = \frac{c_{i,1}+c_{i,2}+\cdots+c_{i,M}}{M}$;
- *deMinW* (*inMinW*): decreasing (increasing) minimum work size $\bar{c_i} = \min_{1\leq k\leq M} c_{i,k}$;
- *deMaxW* (*inMaxW*): decreasing (increasing) maximum work size $\bar{c_i} = \max_{1\leq k\leq M} c_{i,k}$:
- *random*: random ordering.

Then, for each task in the ordered list, we order the processors for mapping its replicas according to PROCMAPCRITERIA, which includes:

- *inE*: increasing energy cost;
- *deR*: decreasing reliability;
- *deP*: decreasing ratio of $-\frac{\log_{10}(1-R(\tau_i,m_k))}{E(\tau_i,m_k)}$ (explained below);
- *random*: random ordering.

Table II: Example

m_k	$E(\tau_i, m_k)$	$R(\tau_i, m_k)$	$1 - R(\tau_i, m_k)$	$\frac{R(\tau_i,m_k)}{E(\tau_i,m_k)}$	$-\frac{\log_{10}(1-R(\tau_i,m_k))}{E(\tau_i,m_k)}$
1	1	0.9	0.1	0.9	1
2	2	0.99	0.01	0.495	1
3	1	0.99	0.01	0.99	2
4	2	0.9	0.1	0.45	0.5

We use the example shown in Table II to explain how to design a better criteria in PROCMAPCRITERIA. Assume there are four processor sets with different energy and reliability configurations. Considering only the reliability, we cannot distinguish between the second and third sets. Apparently, the third set is better since its processors consume less energy and provide the same level of reliability. The problem is the same when ordering processors only according to energy cost. This gives us a hint that we need to consider energy and reliability interactively. A first idea would be to use the ratio $\frac{R(\tau_i,m_k)}{E(\tau_i,m_k)}$, which expresses the reliability per energy unit of task τ_i executing on processor m_k. But consider a task instance with a reliability target $\mathcal{R}_i = 0.98$: it requires either one processor from the second set or two processors from the first set. Both solutions match the reliability goal with the same energy cost 4. We aim at a formula that would give the same weight to both solutions. The ratio $-\frac{\log_{10}(1-R(\tau_i,m_k))}{E(\tau_i,m_k)}$ is a good candidate, because the total energy cost is the sum of all processors while the reliability is a product. This discussion explains how we have derived the third criteria *deP* in PROCMAPCRITERIA, namely to order processors by decreasing ratio of $-\frac{\log_{10}(1-R(\tau_i,m_k))}{E(\tau_i,m_k)}$.

For the mapping phase, we add replicas for task τ_i in the order of the processor list until the reliability target $\mathcal{R}_i$ is reached. The algorithm uses the probability of failure $PoF = 1 - R(\tau_i) = \Pi_{k \in alloc(i)}(1 - R(\tau_i, m_k))$ (Equation 6.4). The mapping process always ensures that: (i) no two replicas of the same task are assigned to the same processor; (ii) the utilization u_k of each processor does not exceed 1.

6.3 Scheduling

In the scheduling phase, we aim at ordering the tasks mapped on each processor, with the objective to minimize the energy consumption during execution. Recall that the success of any replica leads to the immediate cancellation of all the remaining replicas, a crucial source of energy saving. Our approach is to identify a primary replica for each task, then all its other replicas become secondaries. The goal of the proposed scheduling is to avoid overlap between the execution of the primary and secondary replicas for each task: the primary must be terminated as soon as possible, while the secondaries must be delayed as much as possible. Whenever a primary replica of a task succeeds, the energy consumption will be minimal for that task if no secondary replica has started executing yet. Our scheduling algorithm uses a layered approach: first me map the first replica of each task, which we call the primary replica; and then, in a round-robin fashion, we map the remaining replicas (if they exist), which we call the secondaries. Here is a detailed description of Algorithm 8:

1. First we order tasks by criterion TASKSCHEDCRITERIA, for which we propose:
   - *deNR* (*inNR*): decreasing (increasing) number of replicas;
   - *deU* (*inU*): decreasing (increasing) total utilization (sum up the utilization of all replicas);
   - *random*: random ordering.
2. Then we process the list of tasks in that order, and select a primary replica for each task, which we execute as soon as possible on its assigned processor, right after already scheduled primary

Algorithm 7: Replication setting and mapping

Input: A set of tasks τ_i with reliability targets $\mathcal{R}_i$;
a set of heterogeneous processors m_k
Output: An allocation σ_m of all replicas on the processors

begin
  order all the tasks with TASKMAPCRITERIA and renumber them $\tau_1, \ldots, \tau_N$
  /* initialize the utilization of all processors to zero */
  $u \leftarrow [0, \ldots, 0]$
  /* iterate through the ordered list of tasks */
  for $i \in [1, \ldots, N]$ **do**
    /* order processors for each task */
    order all processors for task τ_i with PROCMAPCRITERIA and renumber them $proc_1, \ldots, proc_M$
    /* this ordered list may differ from task to task */
    $k = 1$
    $PoF = 1$
    while $1 - PoF < \mathcal{R}_i$ **do**
      $temp = u_k + u_{i,k}$
      if $temp \leq 1$ **then**
        $u_k = temp$
        $PoF = PoF \times (1 - R(\tau_i, m_k))$
        add one replica of τ_i on $proc_k$
      $k++$
      if $k > m$ **then**
        return *not feasible*
  return σ_m

replicas (if any). We use two different criteria PRIMARYSCHEDCRITERIA for selecting primary replicas:

- *time:* choose the processor that can complete the execution of the replica the earliest (given already made scheduling decisions);
- *energy:* choose the processor that can execute the replica with smallest dynamic energy.

3. Once primary replicas have all been scheduled, we reverse the order of the list of tasks, and we schedule the remaining replicas (considered in a round-robin fashion in the reversed list) as late as possible on their assigned processor. The idea is to mimimize potential overlap between primary and secondaries for each task, hence to delay secondary replicas until the end of the period. The rationale for reverting the task list is that the primary replica of some task τ at the end of the list may have been scheduled after some other primary replica τ', hence the idea to process the secondary replica of τ' before that of τ and push it further away at the end of the period.
4. Finally, there only remains to detail which secondary replica of a task is scheduled first (whenever the task has three replicas or more). We also have two criteria SECONDARYSCHEDCRITERIA for choosing secondary replicas:
   - *time:* choose the replica whose start-up time can be the latest (given already made scheduling decisions); the idea is to minimize overlap by maximizing slack;
   - *energy:* choose the replica whose energy is the highest; the idea is again to minimize overlap, thereby increasing the probability of this costly replica to be cancelled.

As we have two different criteria for both selecting primaries and secondaries, in total, we have four possible combinations, namely *time-time*, *time-energy*, *energy-time* and *energy-energy*. For the baseline scheduling (*randomShuffling*), we randomly order tasks on each processor and execute them in sequence and as early as possible (no idle time until the end of the period).

Algorithm 8: Scheduling

```
Input: An allocation σ_m of all replicas on the processors
Output: An order of execution on each processor
begin
    order all the tasks with TASKSCHEDCRITERIA and renumber them τ_1, ..., τ_N
    /* insert the primary replica for each task at the beginning of each
       processor schedule                                                  */
    for i ∈ [1, ..., N] do
        if PRIMARYSCHEDCRITERIA is "time" then
            schedule the primary replica of task τ_i that could finish at the earliest
        else if PRIMARYSCHEDCRITERIA is"energy" then
            schedule the primary replica of task τ_i that consumes the minimum energy
    /* insert the secondaries backwards from the end of each processor
       schedule                                                            */
    reverse the task ordering
    while there is still at least one replica to be scheduled do
        for i ∈ [1, ..., N] do
            if there is still a replica of τ_i to be scheduled then
                if SECONDARYSCHEDCRITERIA is "time" then
                    schedule the secondary replica of task τ_i that could start the latest
                else if SECONDARYSCHEDCRITERIA is "energy" then
                    schedule the secondary replica of task τ_i that consumes the maximum energy
```

6.4 Lower bound

In this section, we explain how to derive a lower bound for the expected energy consumption of a solution to the optimization problem, namely a mapping/scheduling heuristic that uses some of the selection criteria outlined in Sections 6.2 and 6.3.

For each problem input, namely N tasks τ_i with reliability thresholds $\mathcal{R}_i$, M processors m_k with failure rates λ_k, and with all WCET $c_{i,k}$, we compute a solution, i.e., a mapping and ordering of all replicas. We first use Monte-Carlo simulations (see Section 6.5) and generate several sets of values for the actual execution time $w_{i,k}$ of τ_i on m_k. The values $w_{i,k}$ are drawn *uniformly across processors* as some fraction of their WCET $c_{i,k}$ (refer to Section 6.5.1 for details).

Now, for each set of values $w_{i,k}$, we generate a set of failure scenarios, compute the actual energy consumed for each scenario, and report the average of all these values as the expected energy consumption. A failure scenario operates as follows. We call an event the end of the execution of a task replica on some processor. At each event, we flip a biased coin (weighted with the probability of success of the replica on that processor) to decide whether the replica is successful or not. If it is, we delete all other replicas of the same task. At the end of the execution, we record all the dynamic energy that has been actually spent, accounting for all complete and partial executions of replicas, and we add the static energy given by Equation 6.3. This leads to the energy consumption of the failure scenario. We average the values over all failure scenarios and obtain the expectation, denoted as $E(\{w_{i,k}\})$.

In addition, we also compute a lower bound $LB(\{w_{i,k}\})$ as follows. Our goal is to accurately estimate the energy consumption of an optimal solution. Because the static energy depends upon the subset of processors that are used in the solution (see Equation 6.3), we need to try all possible subsets. Given a processor subset $\mathcal{S}$, we consider each task τ_i independently, and try all possible mappings of replicas of τ_i using only processors in $\mathcal{S}$. Thus we explore all subsets $\mathcal{T}$ of $\mathcal{S}$. A subset $\mathcal{T}$ is *safe* if mapping a replica of τ_i on each processor of $\mathcal{T}$ meets the reliability criteria $\mathcal{R}_i$, and if no strict subset of $\mathcal{T}$ is safe. Note that safe sets are determined using the WCETs $c_{i,k}$, and not using the $w_{i,k}$, because this is part of the problem specification. Now for each safe subset $\mathcal{T}$, we try all possible orderings (there

are $k!$ of them if $|\mathcal{T}| = k$); for each ordering, we compute the expected value of the dynamic energy consumption as follows: if, say, $\mathcal{T} = \{m_1, m_3, m_4\}$ and the ordering is m_3, m_4, m_1, then we compute

$$P_{3,d}w_{i,3} + (1 - e^{-\lambda_3 w_{i,3}})P_{4,d}w_{i,4} + (1 - e^{-\lambda_3 w_{i,3}})(1 - e^{-\lambda_4 w_{i,4}})P_{1,d}w_{i,1}$$

We see that we optimistically assume no overlap between the three replicas, and compute the dynamic energy cost as the energy of the first replica (always spent) plus the energy of the second replica (paid only if the first replica has failed) plus the energy of the third replica (paid only if both the first and second replicas have failed). Note that here we use execution times and failure probabilities based upon the actual execution times $w_{i,k}$ and not upon the WCETs $c_{i,k}$. The value of the sum depends upon the ordering of the processors in $\mathcal{T}$, hence we check the 6 orderings and retain the minimal value. We do this for all safe subsets and retain the minimal value. Finally we sum the results obtained for each task and get the lower bound for the orignal processor subset $\mathcal{S}$. We stress that this bound is not necessarily tight, because our computation assumes no overlap for any replica pair, and does not check the utilization of each processor (which may exceed 1). The final lower bound $LB(\{w_{i,k}\})$ is the minimum over all processor subsets. Although the computation has exponential cost, due to the exploration of all processor subsets $\mathcal{S}$, the computation of the expected energy for a given ordering in a subset $\mathcal{T}$ of $\mathcal{S}$ obeys a closed-form formula.

6.5 Performance evaluation

This section assesses the performance of our different strategies to map and schedule real-time tasks onto heterogeneous platforms. In Section 6.5.1, we describe the parameters and settings used during the experimental campaign. We present the results in Section 6.5.2. The algorithms are implemented in C++ and in R. The related code, data and analysis are publicly available in [49].

6.5.1 Experimental methodology

In the experiments, we have $M = 10$ processors and $N = 20$ tasks which have all the same period $p = 100$. The set of WCETs is generated by the method proposed in [20], as mentioned in Section 6.1.1. The WCET values are controlled by the correlation factor between the different tasks (cor_{task}) and between the different processors (cor_{proc}). These two parameters vary between 0 and 1.[1] For example, $cor_{\text{task}} = 0$ (resp. $cor_{\text{proc}} = 0$) means that the WCET values between different tasks on one processor (resp. between different processors for one task) are completely randomly generated. Inversely, $cor_{\text{task}} = 1$ (resp. $cor_{\text{proc}} = 1$) means that the WCET values between different tasks on one processor (resp. between different processors for one task) are all the same. We also define a parameter *basicWork* as the estimated total utilization of the system with a single replica per task, in order to study the impact of system pressure:

$$basicWork = \frac{\sum_{i,k} c_{i,k}}{M^2 p} = \frac{\sum_{i,k} u_{i,k}}{M^2} \tag{6.5}$$

In Equation 6.5, we use the average WCETs on the M processors ($\frac{\sum_k c_{i,k}}{M}$) to estimate the execution time of task τ_i. We have M processors available during period p, hence *basicWork* represent an estimate of the fraction of time processors are used if each task has a single replica. In the experiments, we vary *basicWork* from 0.1 to 0.3.

[1] We ignored the case when $cor_{\text{task}} = 0$ and $cor_{\text{proc}} = 1$ for the parameter set with big failure rate, because when tasks are completely unrelated, there (very likely) is a task with very long execution time on all processors ($cor_{\text{proc}} = 1$). The number of replicas needed to meet its reliability goal will exceed the number of available processors.

To generate the actual execution times of tasks from their WCETs, we use two parameters. The first one, $\beta_{\text{b/w}}$, is global to all tasks: $\beta_{\text{b/w}}$ is the ratio between the best-case execution time and the worst-case execution time. It is the smallest possible ratio between the actual execution time of a task and its WCET. Therefore, the actual execution time of task τ_i on processor m_k belongs to $[\beta_{\text{b/w}} c_{i,k}, c_{i,k}]$. We consider five possible values of $\beta_{\text{b/w}}$: 0.2, 0.4, 0.6, 0.8, and 1. The second parameter, β_i, is task dependent: β_i describes whether the instance of a task is a small one or a large one. β_i is randomly drawn in $[0, 1]$. A value of $\beta_i = 0$ means that task τ_i has the shortest execution time possible, and $\beta_i = 1$ means that the actual execution is equal to its worst case execution time. Overall, the actual execution time of task τ_i on processor m_k is thus defined as: $w_{i,k} = (\beta_{\text{b/w}} + (1 - \beta_{\text{b/w}})\beta_i)c_{i,k}$.

For a processor m_k in the platform, we fix the static power $P_{k,s}$ at 0.001 as in previous works [111, 113, 115]. But for the dynamic power and the failure rate, we have two sets of parameters. The first set also follows values similar to those of the previous works [111, 113, 115]. For this set, we have a relatively large power and a very small failure rate. Therefore, the replicas using this first set of parameters succeed in almost all cases. Thus, to evaluate our heuristics in the context when failures occur more frequently, we introduce a second set of parameters where the replicas have a smaller power and a larger failure rate. For the first set, we choose randomly the dynamic power $P_{k,d}$ between 0.8 and 1.2, and the failure rate λ_k between 0.0001 and 0.00023. And for the second set, we have $P_{k,d}$ 10 times smaller (between 0.08 and 0.12), and λ_k 100 times larger (between 0.01 and 0.023). With the second set of parameters, the actual reliability of one replica ranges from 0.1 to 0.99. To be more realistic, in our experiments, processors with a larger dynamic power $P_{k,d}$ have a smaller failure rate λ_k. It means that, a more reliable processor costs always more energy than a less reliable one. We guarantee this by ordering inversely the $P_{k,d}$'s and the λ_k's after generating the random values.

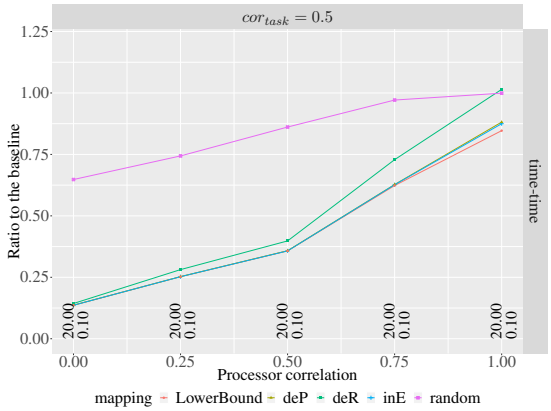
We vary the local reliability target $\mathcal{R}_i$ between 0.9 and 0.98 for the first set and between 0.8 and 0.95 for the second set. This is to give the system a reasonable freedom while mapping and scheduling. The reliability target is relatively high, implying that tasks need plural replicas to reach it. Therefore, we give more tolerance (smaller relialbility threshold) to the second set with a larger failure rate, because otherwise we may not be able to find feasible mappings.

Table III: Ratio of energy consumption of different mapping and scheduling (considering different criteria for ordering tasks) to the baseline method

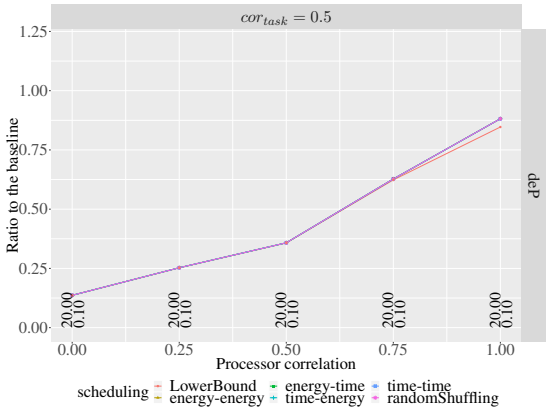
map \ sch	*deNR*	*inNR*	*deU*	*inU*	*random*
deW	0.5655	0.5662	0.5655	0.5660	0.5663
inW	0.5631	0.5635	0.5630	0.5635	0.5635
deMinW	0.5658	0.5662	0.5657	0.5661	0.5665
inMinW	0.5637	0.5642	0.5637	0.5642	0.5641
deMaxW	0.5658	0.5664	0.5657	0.5663	0.5665
inMaxW	0.5629	0.5633	0.5629	0.5633	0.5633
random	0.5633	0.5639	0.5633	0.5638	0.5639

6.5.2 Results

In this section, we analyze the impact of the different parameters on the performance of the heuristics. We choose as default values $\beta_{\text{b/w}} = 1$, *basicWork* $= 0.3$, $\mathcal{R}_i = 0.95$ for the set with big failure rate, and $\mathcal{R}_i = 0.98$ for the set with small failure rate. This set of parameters is chosen to constrain the solution so that we can observe the most interesting results. These trends are representative and more comprehensive set of results are available in [52]. For cor_{task} and cor_{proc}, we fix them at 0.5 as default value. Each experiment is the average of 10 sets of WCET values. For each set, we generate 10 sets



(a) Comparing mapping strategies when using *time-time* as scheduling strategy



(b) Comparing scheduling strategies when using *deP* as mapping strategy

Figure 6.1: Ratio of energy consumption using different mapping and scheduling strategies under small failure rate, when varying cor_{proc}, with $basicWork = 0.3$, $\beta_{\text{b/w}} = 1$, $\mathcal{R}_i = 0.98$ and $cor_{\text{task}} = 0.5$.

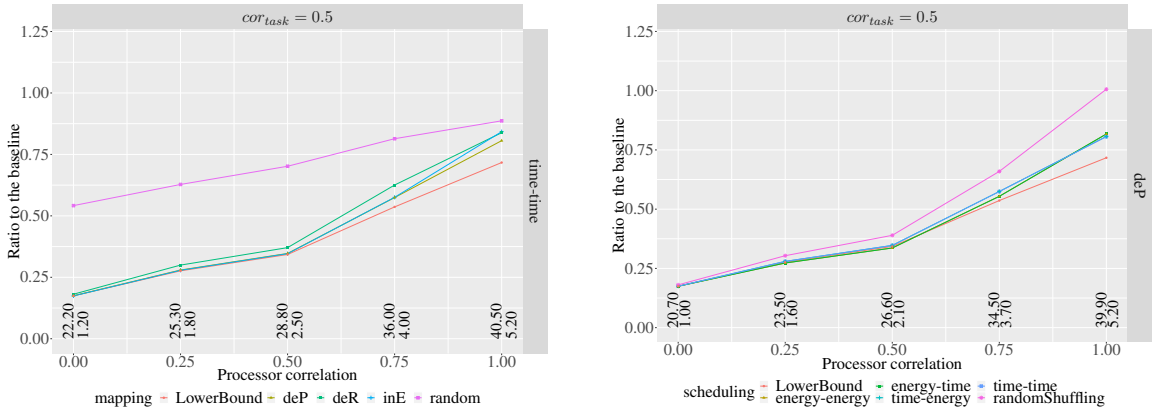
of random $P_{k,d}$ and λ_k values. For each $P_{k,d}$ and λ_k generated, the final result is the average of 10 executions. Overall, we run $1,000$ randomly generated experiments for each set of $\beta_{\text{b/w}}$, $basicWork$, $\mathcal{R}_i$, cor_{task} and cor_{proc} values. The total number of experiments ran is 3,075,000 for each heuristic. Each result is represented as a ratio to the random baseline method which is defined as follows: for each task, we add replicas randomly on available processors until reaching its reliability target during the mapping phase; for scheduling, we randomly order replicas mapped on each processor and execute them in sequence and as soon as possible. We also compare different strategies with the lower bound proposed in Section 6.4. We report on these figures the average number of replicas needed in total for 20 tasks (on the left side) and of failures that occur for the $1,000$ random trials for each setting (on the right side). These numbers are reported in black above the horizontal axis in each figure.

Ordering tasks for mapping and scheduling

In Table III, we calculated the ratio of combinations of different mapping and scheduling methods to the baseline method, when considering different criteria for ordering tasks. We can see that, in the whole set of experiments, all criteria for ordering tasks perform equally well (around 56%). The difference between the best and the worst performance is only around 0.36%. Hence these criteria do not critically influence energy consumption. In the following results, for the task ordering, we only consider the decreasing average WCET (*deW*) for the mapping, and the decreasing utilization (*deU*) for the scheduling, which give priority to the tasks that putting more pressure to the system. We then focus on selecting processors during the mapping phase, and on choosing primary and secondary replicas during the scheduling phase.

Processor correlation

Figures 6.1 and 6.2 show results when processor correlation varies, under small and big failure rate respectively. We found that our strategies consume less than 25% of the energy needed by the baseline strategy when $cor_{\text{proc}} = 0$, and the result is close to the lower bound. But we can observe that this percentage increases with cor_{proc}.



(a) Comparing mapping strategies when using *time-time* as scheduling strategy

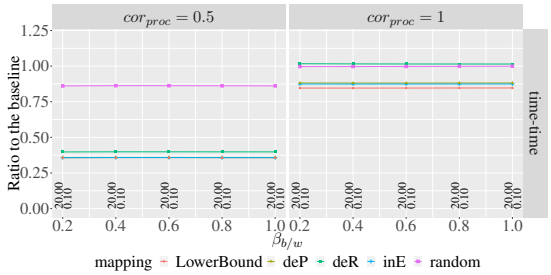
(b) Comparing scheduling strategies when using *deP* as mapping strategy

Figure 6.2: Ratio of energy consumption using different mapping and scheduling strategies under big failure rate, when varying cor_{proc}, with $basicWork = 0.3$, $\beta_{\text{b/w}} = 1$, $\mathcal{R}_i = 0.95$ and $cor_{\text{task}} = 0.5$.

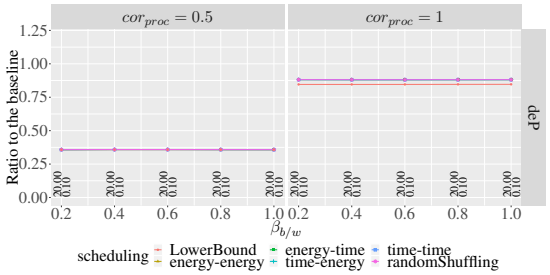
For the mapping phase, in Figure 6.1a we observe that, in the small failure rate case, ordering the processors by decreasing reliability (*deR*) has a worse performance than ordering them by increasing energy cost (*inE*) or decreasing proportion (*deP*) in nearly all cases. This is reasonable because, in this case, the reliability of all tasks on all processors is relatively high and only one replica per task is enough to meet its reliability threshold. Therefore, we cannot have significant improvements by using more reliable processors, but these can cost much more energy. Furthermore, *inE* and *deP* perform similarly and have a performance close to that of the lower bound, except when $cor_{\text{proc}} \approx 1$. In the big failure rate case (Figure 6.2a), we see that, when cor_{proc} is not equal to 1, *inE* performs better than *deR*. But when cor_{proc} increases to 1, the performance of *deR* catches up. And in all cases, *deP* performs better than, or similarly to the best strategy between *deR* and *inE*.

And for the scheduling strategies, Figures 6.1b and 6.2b show that, there is little difference between our different criteria, the random one excepted. But we can still observe more difference in the big failure rate case. With $cor_{\text{proc}} \neq 1$, we have *energy* criteria for primary replica choosing slightly better than *time*, and when $cor_{\text{proc}} = 1$, the *energy* becomes worse than the *time*.

This is because, for most of the cases, the reliability of our replica is high, so that we can simply choose the replica which costs the least energy as primary, and delete all secondary when it finishes successfully. But in the case of $cor_{\text{proc}} = 1$, the WCETs of each task on different processors are the same, so the order of the processors for any task is the same, and is relative to the power and reliability parameters ($P_{k,d}$ and λ_k). This can result in a few fully used processors, with the other processors being empty. Also, for *time* criteria, primary replicas will be randomly balanced on different fully used processors, because every replica of a task has the same WCET. But for *energy*, the processors which cost less energy are the same for all tasks. Then these processors will execute all mapped replicas as primary, and others will execute all mapped replicas as secondary, which increases the overlap. This is why, when $cor_{\text{proc}} = 1$, we cannot save as much energy as in other cases, and this is why the *energy* criteria performs worse than the *time* criteria.

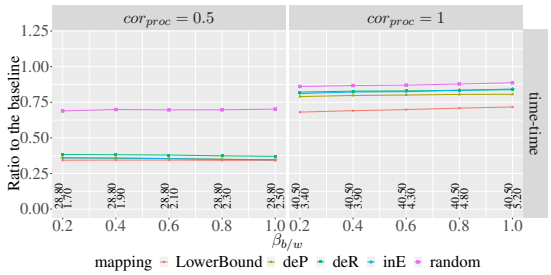


(a) Comparing mapping strategies when using *time-time* as scheduling strategy

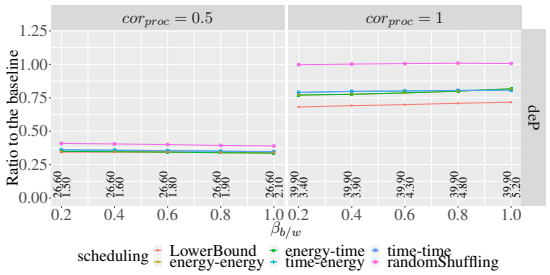


(b) Comparing scheduling strategies when using *deP* as mapping strategy

Figure 6.3: Ratio of energy consumption using different mapping and scheduling strategies under small failure rate when varying $\beta_{\text{b/w}}$, with $basicWork = 0.3$, $\mathcal{R}_i = 0.98$ and $cor_{\text{task}} = 0.5$.



(a) Comparing mapping strategies when using *time-time* as scheduling strategy

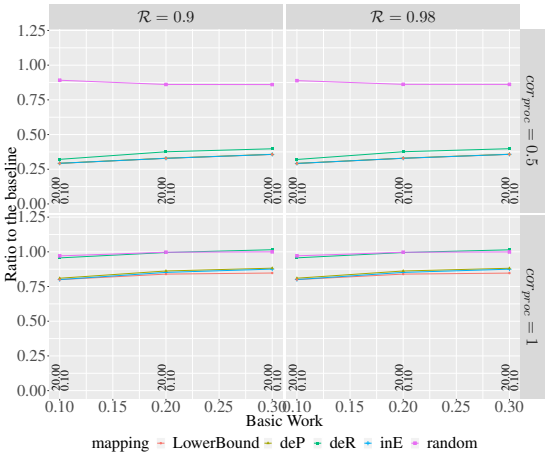


(b) Comparing scheduling strategies when using *deP* as mapping strategy

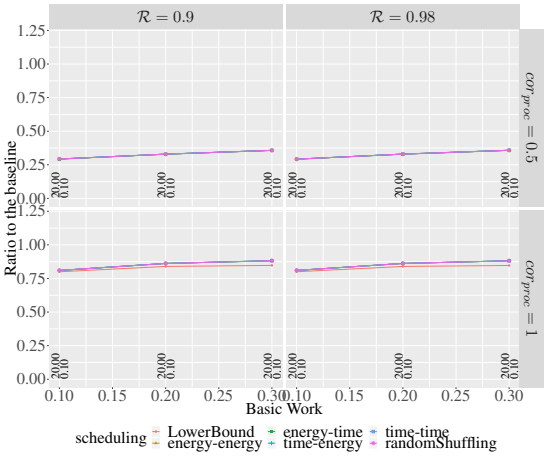
Figure 6.4: Ratio of energy consumption using different mapping and scheduling strategies under big failure rate when varying $\beta_{\text{b/w}}$, with $basicWork = 0.3$, $\mathcal{R}_i = 0.95$ and $cor_{\text{task}} = 0.5$.

Task variability

Figures 6.3 and 6.4 present the results with small and big failure rates when $\beta_{\text{b/w}}$ varies. We observe that, for almost all the mapping and scheduling criteria, the results are similar whatever the value of $\beta_{\text{b/w}}$. This is because we map and schedule tasks based on their WCETs, so the mapping and scheduling results are independent of the value of $\beta_{\text{b/w}}$. Furthermore, each task i has the same β_i on the different processors. Therefore the energy consumption ratios tend to be similar. But in the case of $cor_{\text{proc}} \approx 1$, we can see from Figure 6.4b (under big failure rate where more than one replica per task needed) that the ratio of all scheduling criteria increases slightly with $\beta_{\text{b/w}}$, and the two strategies using $energy$ as primary replica choosing criteria increase faster than the two others which use $time$. In fact, when we have a larger value of $\beta_{\text{b/w}}$, the actual execution time is closer to the WCET. So that, although the mappings and schedulings are the same for different $\beta_{\text{b/w}}$, replicas will take longer time during the actual execution. At the same time, as explained in the previous paragraph, we have a more serious overlap in the case of $cor_{\text{proc}} = 1$ and for $energy$ criteria. This is why $cor_{\text{proc}} = 1$ performs differently when varying $\beta_{\text{b/w}}$. However this phenomenon is not obvious ($\approx 5\%$), so we can always conclude that the result is independent of $\beta_{\text{b/w}}$. When $\beta_{\text{b/w}}$ is small, actual execution times can greatly differ from the WCETs used for mapping and scheduling. However, in this case, our heuristics have a performance similar to that of the lower bound, which shows that they are very robust.



(a) Comparing mapping strategies when using *time-time* as scheduling strategy



(b) Comparing scheduling strategies when using *deP* as mapping strategy

Figure 6.5: Ratio of energy consumption using different mapping and scheduling strategies when varying *basicWork* and $\mathcal{R}_i$, under small failure rate, with $\beta_{\text{b/w}} = 1$ and $cor_{\text{task}} = 0.5$.

Utilization and reliability threshold

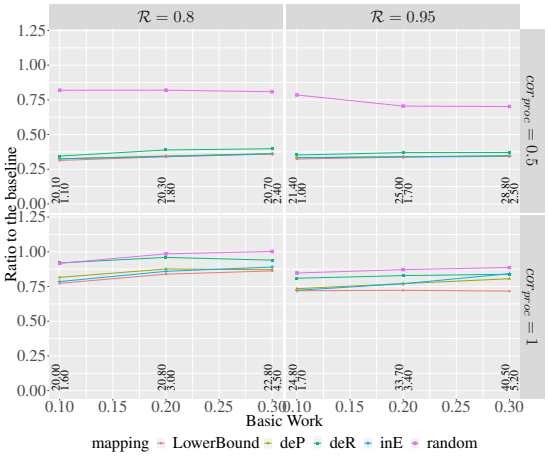
From Figures 6.5 and 6.6, we observe the performance of different mapping and scheduling criteria when varying *basicWork* and $\mathcal{R}_i$.

We remark that the results are similar under big and small failure rates when we have cor_{proc} not close to 1. For example, we can see the case $cor_{\text{proc}} = 0.5$ in the first row of Figures 6.5 and 6.6. During the mapping phase, *deR* has a slightly worse performance than *inE* and *deP*, which perform similarly to the lower bound. During the scheduling phase, all criteria have similar performance, including the random strategy and the lower bound.

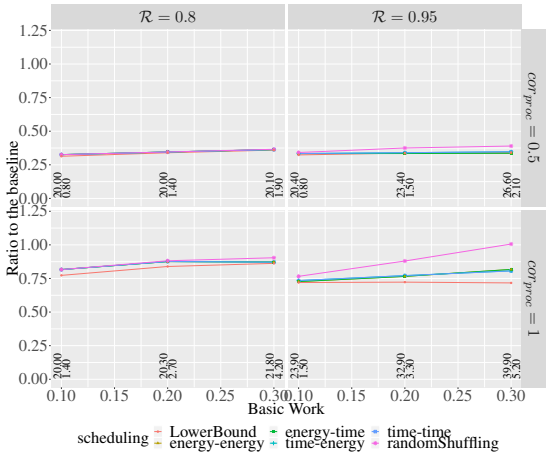
On the contrary, with $cor_{\text{proc}} \approx 1$, the heuristics perform differently with big and low failure rates. For the small failure rate, during the mapping phase, we see from the second row of Figure 6.5a that *deR* is worse than *inE* and *deP* and has a performance similar to the random case, while *inE* and *deP* provide results similar to the lower bound. During the scheduling phase, in the second line of Figure 6.5b we can still see similar performance for all criteria. For the big failure rate case, during the mapping phase, the second row of Figure 6.6a shows that the difference between different criteria becomes smaller when *basicWork* and $\mathcal{R}_i$ increase. Inversely, the difference with the lower bound becomes larger, but it is still less than 10% (except random), even in the worst case. For small values of *basicWork* and $\mathcal{R}_i$, *deR* performs worse, but when *basicWork* and $\mathcal{R}_i$ increase, *deR* becomes similar to *inE*, or even better. *deP* has always better or similar performance than *deR* and *inE*. For the scheduling phase, from the second row of Figure 6.6b, we can still find similar performance on all our criteria, but their difference with the lower bound increases to 10% as in the mapping phase. The reason is that with the increase of system load, it becomes harder to map all replicas to their best processors, while the lower bound is calculated without considering utilization constraints.

Number of failures

We counted the number of replicas that failed during the execution in each experiment. In the set with small failure rate, we have on average 0.44% failed replicas. Thus, in most of the cases, it is enough to



(a) Comparing mapping strategies when using *time-time* as scheduling strategy



(b) Comparing scheduling strategies when using *deP* as mapping strategy

Figure 6.6: Ratio of energy consumption using different mapping and scheduling strategies when varying *basicWork* and $\mathcal{R}_i$, under big failure rate, with $\beta_{\text{b/w}} = 1$ and $cor_{\text{task}} = 0.5$.

have a single replica mapped on the processor that costs the least energy, because failures are scarce. On the contrary, in the set with big failure rate, the average rate of failed replicas increases to 7.57%. We can observe that the *deP* mapping method used in conjunction with $time$ criterion for choosing primary replica when processors are highly correlated, or with $energy$ criterion in other cases achieves the best performance, or a performance similar to the best observed one, in both cases. This confirms that the performance of our best heuristics is not affected by the failure rate.

Success rate

All the tested heuristics were able to find a valid solution in all tested configurations with small failure rate. And in the big failure rate cases, heuristics were able to build valid solutions for more than 99.94% of the instances. The very high success rate of our experiments shows the robustness of our approach.

Summary

In conclusion, our strategies can save more than 40% of the energy consumed by the baseline, except in the high processor correlation case. The ratio to the baseline can be as low as 20% in the best case.

As for the different criteria used in the heuristics, we find that the *deP* method is the best processor ordering during the mapping phase.

For scheduling, we can find from the result that, all our primary-secondary choosing criteria have a similar result as the lower bound, except a difference of 10% in the case of $cor_{\text{proc}} \approx 1$. This means that our primary-secondary choosing heuristic performs well. On the other hand, we point out that strategies with the same method for choosing the primary replica but different methods for choosing secondary replicas, perform similarly. Hence the strategy for choosing the primary replica has much more impact than the one for choosing secondary replicas. We can find that, *time-time* and *time-energy* criteria performs better when $cor_{\text{proc}} \approx 1$, and *energy-energy* and *energy-time* have better performance in other cases.

The performance of these best heuristics is only 17.0% higher than the lower bound in the worst case. Furthermore, we report a median value only 3.5% higher than that of the lower bound; and the average value is only 4.3% higher. We can confidently conclude that our best strategies perform remarkably well over the whole experimental setting.

6.6 Conclusion

In this work, we have studied the problem of executing periodic real-time tasks on an heterogeneous platform, with several objectives: minimizing the energy consumption, guaranteeing some reliability thresholds, and meeting all deadlines. For each task, we decide how many replicas should be launched, and on which processors to map them. We tagged one replica per task as "primary" replica and the other ones as "secondary" replicas. To obtain an absolute measure for the evaluation of our heuristics, we have computed a theoretical lower bound on energy consumption. Extensive simulations show that our best heuristic always achieve very good performance, very close to the lower bound (on average only 4% higher than this lower bound). This performance was reached by considering processors in the *deP* order when mapping the replicas of a task (roughly speaking, *deP* is the ratio of a task failure rate by its energy cost), by executing primary replicas as soon as possible and secondary ones as late as possible, and by tagging replicas as "primary" using an earliest completion time criterion when processors are highly correlated, using an smallest energy criterion otherwise. Furthermore, while all decisions are taken with the worst-case execution times (WCETs) of tasks as only input, the simulations used the actual execution times; the best heuristic always achieved excellent performance even when the actual execution times were far smaller than the WCETs, showing the robustness of our approach.

Future work will aim at extending the algorithms to periodic graphs of tasks instead of independent task sets. The dependences between tasks will dramatically complicate the mapping and scheduling problems.

Conclusion

In this thesis, we have studied two challenging problems, namely, resilience and energy efficiency, that must be addressed to cope with future Exascale platforms. More precisely, we designed new scheduling strategies to take multiple different objectives into account (e.g., minimizing execution time, achieving predefined reliability goal, minimizing energy consumption). Given a multi-criteria optimization problem, we first studied its complexity. Either we proved its NP-completeness and designed efficient polynomial-time heuristics, or we proposed optimal solutions (for certain steps). Then, we used extensive simulations to evaluate the effectiveness of our methods, during which we compared with the state-of-the-art results whenever there exist, or we analytically computed the lower bound as a reference.

There are two parts in this manuscript, where we studied workflow applications and real-time applications respectively. In Part I, we focused on scheduling workflows (task graphs) onto a parallel platform, which is subject to fail-stop errors. Checkpointing is applied to cope with failures, but it also introduces redundancy in time. Obviously, in order to minimize the total execution time, the frequency of checkpointing (putting checkpoints after which tasks) should be carefully designed, which is supposed to depend on failure rate, task execution time and checkpointing cost etc. By studying the actual executions (see examples in Sections 2.1 and 3.1), we identified 'processor interference' as the main source of massive re-execution in case of a failure. It is because a failure on one processor will cause task re-execution on other processors, during which another failure may occur. This may lead to infinite execution time if there is no appropriate checkpointing strategies. Based on this observation, we succeeded in designing optimal strategies for M-SPGs and general-purpose heuristics for arbitrary workflows. The main work and contributions are summarized in the following two paragraphs:

Optimal solutions for special classes of task graphs (Chapter 2)

In the first step, we took advantage of the recursive structure of M-SPGs and scheduled them as a set of superchains using proportional mapping heuristic, which enabled us to optimally checkpoint inside each of the super chain given that all exit tasks are checkpointed. We proposed a novel idea of 'task checkpoints' that save not only the output files of the checkpointed task, but also unsaved files in local memory that will be used afterwards. Moreover, our method enables us to evaluate the makespan by sampling the expected execution time of each task individually and by Monte-Carlo simulations. For CkptNone strategy, when applied to general DAGs, we have established that the problem of computing the expected makespan is #P-complete.

Generic approaches for arbitrary task graphs (Chapter 3)

Following the previous results, we extended our methods to general task graphs. Besides checkpointing all the crossover dependencies in order to avoid processor interference, we identified 'induced tasks', by checkpointing which we could take advantage of possible idle times and isolate a sequence of tasks without waiting time. This enabled us to properly reuse the dynamic programming proposed in Chapter 2

to optimally put checkpoints inside each sequence of tasks. Moreover, we provided a loose checkpointing strategy in case that the failures are rare. We revisited the classical list scheduling algorithms and provided their variations in order to reduce the crossover dependencies, thus checkpointing cost. To evaluate the makespan, we designed a discrete-event simulator. Extensive simulations showed that our strategies achieved an efficient trade-off between CKPTALL and CKPTNONE. To the best of our knowledge, we proposed the first scheduling and checkpointing strategies for general workflows beyond a chain of tasks.

In Part II, we studied real-time task allocation onto homogeneous/heterogeneous platforms that are subject to transient faults. Replication is considered to ensure a prescribed reliability threshold, which also introduces resources redundancy that may lead to huge energy consumption and/or exceeding deadlines. We do not want to add replicas at any cost (in terms of energy consumption), nor miss any deadline, because timeliness is as important as correctness for real-time systems. Therefore, we faced a complicated tri-optimization problem. We identified overlaps between replicas as a main source of energy consumption. It is because once a task replica succeed, we could safely interrupt the current executing replicas of the task or completely cancel those have not been launched yet. By simultaneously considering the three criteria (i.e., deadlines, reliability and energy), we designed heuristics to decide the number of replicas for each task, to map and schedule replicas onto processors, in order to meet all deadlines and the predefined reliability threshold, while minimizing energy consumption by reducing overlaps between replicas. We also proved several complexity results for the sole scheduling phase. The main contributions are summarized as below:

Homogeneous platforms (Chapter 5)

This chapter is dedicated to homogeneous platforms, which means that we have identical processors. This gave an advantage that we could pre-compute the number of replicas needed for each task at each frequency level to meet its reliability goal. We improved the state-of-the-art approach through three steps: 1) we used a more realistic formula for expected energy consumption at the replica setting phase; 2) we used a layered WFD mapping heuristic in order to spread primaries and secondaries over processors and lead to a more balanced load; and 3) we designed several scheduling heuristics to start primary replicas as soon as possible and to delay secondary replicas to minimize overlapping, which is achieved by re-ordering chunks between intervals, pre-fetching available primary replicas and taking advantages of idle times. Extensive experiments conducted for various range of parameters shown that our best scheduling heuristics significantly outperform the competitor with an average gain in energy of 20%.

Heterogeneous platforms (Chapter 6)

Here, we faced a more complicated problem than in the previous chapter, as each processor has different parameters (i.e., speed, failure rate, power). The set of processors chosen for each task matters, no only for minimizing overlapping, but also because each processor performs differently for each single task. In other words, a processor achieves different levels of reliability and consumes different amount of energy for different tasks. We were able to conclude that: 1) in which order we consider processors in the mapping phase and 2) how to choose primary replica and secondary replicas in the scheduling phase are the two most important criteria for minimizing energy. To assess the performance of proposed heuristics, besides comparing to the random baseline, we computed a theoretical lower bound and launched exhaustive experiments with comprehensive set of parameters. Results showed that our best heuristic always achieves a performance that is far better than the baseline (with a gain of 40%), and is very close to the lower bound (with an average value only 4% higher than the lower bound).

Perspectives and future work

The work conducted in this thesis can be pursued in multiple directions. Throughout this thesis, at the end of each chapter, we have pointed out several interesting future directions. We review here some short-term and long-term perspectives.

Short-term perspectives

In short term, we plan to improve the identified weaknesses in our current work, as well as extend our solutions to different types of applications or to more general models. We list below several promising topics:

- In the work of Part I, we focused on a DAG of atomic tasks. Extending our approach to a DAG of parallel tasks would be an interesting topic. Such an extension raises yet another significant challenge: the number of processors assigned to each task becomes a parameter to the proposed solutions, which will cause a dramatic impact on both performance and resilience; in particular, changing the processor number will also change the probability of failure;

- In the study of homogeneous real-time platforms (Chapter 5), we were able to pre-compute replica sets for different frequency levels. But the actual energy consumption of the most energy efficient replica configuration is not necessarily minimal. For example, a lower frequency with more replicas may lead to unavoidable overlapping or may use more processors that consume more static energy. We should further study the influences of different replica configurations on the total number of processors needed and on the optimization steps afterwards. Moreover, the formula for calculating expected energy consumption at the first step could be further refined, as now we pessimistically assumed that all replicas are executed sequentially;

- For real-time heterogeneous platforms studied in Chapter 6, we could extend our strategies to processors with multiple frequency levels and/or a set of tasks with multiple deadlines. Different deadlines complicate the problem, as we have to take into consideration task preemption, instead of simple permutation of tasks in the same interval. This will call for a different scheduling strategy, e.g., EDF, as well as advanced approaches for minimizing overlaps;

- It could also be interesting to implement and test our methods in practice. But the difficulty would be that it is not easy to control the failure rate on real machines, thus to verify the effectiveness of our approaches under different scenarios.

Long-term perspectives: failure-aware DAG scheduling

For workflows, we would like to study failure-aware DAG scheduling, with could enable us to simultaneously consider the scheduling and checkpointing phases. As already pointed out, this would give a chance to global improvements, but we are lacking techniques for the moment. Extending the branch and bound methods reviewed in Section 1.2.3 for scheduling DAGs would be a possible way. A natural idea would be incorporating checkpoint decisions into the evaluation function to estimate the makespan that can be achieved when exploring a solution sub-tree. However, to achieve a satisfactory result, our first step would be finding an accurate evaluation function to prune solution sub-trees; in other words, we have to answer how to analytically estimate the expected execution time of a complete solution. This is the core to apply the branch and bound method, as the approach requires a lower-bound on the objective achievable by any solution in the considered branch. The second question to answer would be the

scalability of the approach since the problem is more combinatorial with checkpoints than it was. What is limitation on the size of graphs we could tackle? Overall, this is a challenging proposition, and we hope to get some new breakthroughs.

Long-term perspectives: energy-aware real-time workflow allocation

Considering real-time workflows instead of independent task sets is an interesting direction, and the dependencies between tasks will further complicate the problem. As already pointed out, the overlap between replicas is a main source for energy consumption. We would like to dig into workflow scheduling techniques aiming at reducing overlaps between replicas of the same task while meeting all deadlines. From what we have seen from literature, the common assumption for real-time workflows is that all the tasks in one workflow instance have the same deadline, which gives more flexibility when scheduling tasks. Also, there is a work considering the same problem but for a DAG. In [114] (Chapter 3), the authors consider cost minimization (which can be energy minimization) when scheduling a DAG under deadline and reliability constraints. In this work, because of the dependency between tasks and the as-soon-as-possible scheduling, their solution tends to schedule the different replicas of a single task simultaneously. This can lead to a significant waste of energy in our context. Therefore, designing a real-time workflow allocation method, which obeys all dependencies and deadlines and maximizing slots between primary replica and secondary replicas of a same task instance, will be our first step to extend our work to real-time workflow applications.

Bibliography

[1] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. “Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids.” In: *1st ACM SWEET SIGMOD*. ACM. 2012.

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. “Kepler: an extensible system for design and execution of scientific workflows.” In: *Proc. of 16th SSDBM*. IEEE, 2004, pp. 423–424.

[3] M. Ansari, S. Safari, A. Y. Khaksar, M. Salehi, and A. Ejlali. “Peak Power Management to Meet Thermal Design Power in Fault-Tolerant Embedded Systems.” In: *IEEE Transactions on Parallel and Distributed Systems* (2018).

[4] I. Assayad, A. Girault, and H. Kalla. “A Bi-Criteria Scheduling Heuristic for Distributed Embedded Systems under Reliability and Real-Time Constraints.” In: *Dependable Systems Networks (DSN)*. IEEE, 2004.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. “StarPU: a unified platform for task scheduling on heterogeneous multicore architectures.” In: *Concur. and Comp.: Pract. and Exp.* 23.2 (2011), pp. 187–198.

[6] G. Aupy, A. Benoit, H. Casanova, and Y. Robert. “Scheduling computational workflows on failure-prone platforms.” In: *Int. J. of Networking and Computing* 6.1 (2016), pp. 2–26.

[7] H. Aydin and Q. Yang. “Energy-Aware Partitioning for Multiprocessor Real-Time Systems.” In: *Proc. 17th Int. Symp. Parallel and Distributed Processing*. IPDPS ’03. IEEE Computer Society, 2003.

[8] R. Baldoni, J. .-.-. Helary, A. Mostefaoui, and M. Raynal. “A communication-induced checkpointing protocol that ensures rollback-dependency trackability.” In: *Proc. IEEE 27th International Symposium on Fault Tolerant Computing*. IEEE, 1997, pp. 68–77.

[9] L. Bautista Gomez and F. Cappello. “Detecting and Correcting Data Corruption in Stencil Applications through Multivariate Interpolation.” In: *FTS*. IEEE, 2015.

[10] L. Bautista Gomez and F. Cappello. “Detecting Silent Data Corruption Through Data Dynamic Monitoring for Scientific Applications.” In: *SIGPLAN Notices* 49.8 (2014), pp. 381–382.

[11] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. “Assessing general-purpose algorithms to cope with fail-stop and silent errors.” In: *ACM Trans. Parallel Computing* 3.2 (2016).

[12] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello. “Lightweight Silent Data Corruption Detection Based on Runtime Data Analysis for HPC Applications.” In: *HPDC*. ACM, 2015.

[13] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. “Characterization of scientific workflows.” In: *Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 2008, pp. 1–10.

[14] E. Bini and G. C. Buttazzo. “Measuring the performance of schedulability tests.” In: *Real-Time Systems* 30.1-2 (2005), pp. 129–154.

[15] H. L. Bodlaender and B. de Fluiter. “Parallel algorithms for series parallel graphs.” In: *Algorithms - ESA’96*. Springer, 1996, pp. 277–289.

[16] H. L. Bodlaender and T. Wolle. "A Note on the Complexity of Network Reliability Problems." In: *IEEE Trans. Inf. Theory* 47 (2004), pp. 1971–1988.

[17] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. "Algorithm-based fault tolerance applied to high performance computing." In: *J. Parallel Distrib. Comput.* 69.4 (2009), pp. 410–416.

[18] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, et al. "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems." In: *Journal of Parallel and Distributed computing* 61.6 (2001), pp. 810–837.

[19] L. C. Canon and E. Jeannot. "Correlation-Aware Heuristics for Evaluating the Distribution of the Longest Path Length of a DAG with Random Weights." In: *IEEE Trans. Parallel Distributed Systems* (2016).

[20] L.-C. Canon, M. El Sayah, and P.-C. Héam. "A markov chain monte carlo approach to cost matrix generation for scheduling performance evaluation." In: *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2018, pp. 460–467.

[21] C. Cao, T. Herault, G. Bosilca, and J. Dongarra. "Design for a Soft Error Resilient Dynamic Task-Based Runtime." In: *IPDPS*. IEEE, 2015, pp. 765–774.

[22] K. Cao, G. Xu, J. Zhou, T. Wei, M. Chen, and S. Hu. "QoS-Adaptive Approximate Real-Time Computation for Mobility-Aware IoT Lifetime Optimization." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018), pp. 1–1. ISSN: 0278-0070. DOI: 10.1109/TCAD.2018.2873239.

[23] K. Cao, J. Zhou, P. Cong, L. Li, T. Wei, M. Chen, S. Hu, and X. S. Hu. "Affinity-driven modeling and scheduling for makespan optimization in heterogeneous multiprocessor systems." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).

[24] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. "Toward Exascale Resilience: 2014 update." In: *Supercomputing frontiers and innovations* 1.1 (2014).

[25] H. Casanova, J. Herrmann, and Y. Robert. "Computing the expected makespan of task graphs in the presence of silent errors." In: *P2S2'2016, the 9th Int. Workshop on Programming Models and Systems Software for High-End Computing*. IEEE Press, 2016.

[26] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. "Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines." In: *Scientific Programming* 5.3 (1996), pp. 173–184.

[27] M. Chtepen, F. H. A. Claeys, B. Dhoedt, F. D. Turck, P. Demeester, and P. A. Vanrolleghem. "Adaptive Task Checkpointing and Replication: Toward Efficient Fault-Tolerant Grids." In: *IEEE Trans. Parallel Distributed Systems* 20.2 (2009), pp. 180–190.

[28] E. G. Coffman Jr., M. R. Garey, and D. S. Johnson. "Approximation Algorithms for Bin Packing: A Survey." In: *Approximation Algorithms for NP-hard Problems*. Ed. by D. S. Hochbaum. PWS Publishing Co., 1997, pp. 46–93.

[29] A. Darte, Y. Robert, and F. Vivien. *Scheduling and automatic parallelization*. Birkhäuser, 2000. ISBN: 978-3-7643-4149-7.

[30] R. I. Davis and A. Burns. "A Survey of Hard Real-time Scheduling for Multiprocessor Systems." In: *ACM Comput. Surv.* 43.4 (Oct. 2011), 35:1–35:44. ISSN: 0360-0300. DOI: 10.1145/1978802.1978814. URL: http://doi.acm.org/10.1145/1978802.1978814.

[31] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, et al. “Pegasus: A framework for mapping complex scientific workflows onto distributed systems.” In: *Scientific Programming* 13.3 (2005), pp. 219–237.

[32] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger. “Pegasus, a Workflow Management System for Science Automation.” In: *Future Generation Computer Systems* 46 (2015), pp. 17–35.

[33] A. Dixit and A. Wood. “The impact of new technology on soft error rates.” In: *2011 International Reliability Physics Symposium*. IEEE. 2011, 5B–4.

[34] J. Dıaz, J. Petit, and M. Serna. “A survey of graph layout problems.” In: *ACM Computing Surveys* 34.3 (2002), pp. 313–356.

[35] A. B. Downey. “The structural cause of file size distributions.” In: *MASCOTS 2001*. IEEE. 2001, pp. 361–370.

[36] M. Drozdowski. *Scheduling for Parallel Processing*. Computer Communications and Networks. Springer, 2009.

[37] R. Duan, R. Prodan, and T. Fahringer. “Dee: A distributed fault tolerant workflow enactment engine for grid computing.” In: *International Conference on High Performance Computing and Communications*. Springer. 2005, pp. 704–716.

[38] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, et al. “Askalon: A development and grid computing environment for scientific workflows.” In: *Workflows for e-Science*. Springer, 2007, pp. 450–471.

[39] X. Fan, C. S. Ellis, and A. R. Lebeck. “The synergy between power-aware memory systems and processor voltage scaling.” In: *International Workshop on Power-Aware Computer Systems*. Springer. 2003, pp. 164–179.

[40] *Frequency Behavior-Intel*. https://en.wikichip.org/wiki/intel/frequency_behavior.

[41] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[42] A. Girault and H. Kalla. “A novel bicriteria scheduling heuristics providing a guaranteed global system failure rate.” In: *IEEE Trans. Dependable and Secure Computing* 6.4 (2009), pp. 241–254.

[43] H. H. Greenberg. “A Branch-Bound Solution to the General Scheduling Problem.” In: *Oper. Res.* 16.2 (1968), pp. 353–361.

[44] T. Guo, J. Liu, W. Hu, and M. Wei. “Energy-Aware Fault-Tolerant Scheduling Under Reliability and Time Constraints in Heterogeneous Systems.” In: *Intelligent Computing Methodologies*. Ed. by D.-S. Huang, M. M. Gromiha, K. Han, and A. Hussain. Springer, 2018, pp. 36–46.

[45] Y. Guo, D. Zhu, H. Aydin, J.-J. Han, and L. T. Yang. “Exploiting primary/backup mechanism for energy efficiency in dependable real-time systems.” In: *Journal of Systems Architecture* 78 (2017), pp. 68–80. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2017.06.008. URL: http://www.sciencedirect.com/science/article/pii/S1383762116302624.

[46] J. N. Hagstrom. “Computational complexity of PERT problems.” In: *Networks* 18.2 (1988), pp. 139–147.

[47] L. Han. *Checkpointing Workflows: Simulation Code*. https://doi.org/10.6084/m9.figshare.5057650.v3. 2017.

[48] J.-J. Han, W. Cai, and D. Zhu. "Resource-aware Partitioned Scheduling for Heterogeneous Multicore Real-time Systems." In: *Proceedings of the 55th Annual Design Automation Conference*. DAC '18. San Francisco, California: ACM, 2018, 124:1–124:6. ISBN: 978-1-4503-5700-5. DOI: 10.1145/3195970.3196103. URL: http://doi.acm.org/10.1145/3195970.3196103.

[49] L. HAN. "Heterogeneous real-time systems." In: (Mar. 2020). DOI: 10.6084/m9.figshare.11925423.v1. URL: https://figshare.com/articles/Heterogeneous_real-time_systems/11925423.

[50] L. Han, L.-C. Canon, H. Casanova, Y. Robert, and F. Vivien. *Checkpointing Workflows for Fail-Stop Errors*. Research Report 9068. Short version appears in the proceedings of the IEEE Cluster Conference, 2017 Extended version will appear in IEEE Trans. Computers, 2018. INRIA, May 2017.

[51] L. Han, L.-C. Canon, J. Liu, f. vivien frédéric, and Y. Robert. *Code to schedule for periodic real-time tasks under reliability constraints with minimal energy consumption*. Sept. 2019. DOI: 10.6084/m9.figshare.9778319.v1. URL: https://figshare.com/articles/Code_to_schedule_for_periodic_real-time_tasks_under_reliability_constraints_with_minimal_energy_consumption/9778319/1.

[52] L. Han, Y. Gao, J. Liu, Y. Robert, and F. Vivien. *Energy-aware strategies for reliability-oriented real-time task allocation on heterogeneous platforms*. Research report 92324. INRIA, Feb. 2020.

[53] Q. Han. "Energy-aware Fault-tolerant Scheduling for Hard Real-time Systems." PhD thesis. Florida International University, 2015. DOI: 10.25148/etd.FIDC000077.

[54] M. A. Haque, H. Aydin, and D. Zhu. "On reliability management of energy-aware real-time systems through task replication." In: *IEEE Transactions on Parallel and Distributed Systems* 28.3 (2017), pp. 813–825.

[55] K. Hashimoto, T. Tsuchiya, and T. Kikuno. "Fault-secure scheduling of arbitrary task graphs to multiprocessor systems." In: *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*. 2000, pp. 203–212. DOI: 10.1109/ICDSN.2000.857536.

[56] T. Hérault and Y. Robert, eds. *Fault-Tolerance Techniques for High-Performance Computing*. Computer Communications and Networks. Springer Verlag, 2015.

[57] U. Hönig and W. Schiffmann. "A Parallel Branch–and–Bound Algorithm for Computing Optimal Task Graph Schedules." In: *Grid and Cooperative Computing: Second International Workshop, GCC 2003, Shanghai, China, December 7-10, 2003, Revised Papers, Part II*. Ed. by M. Li, X.-H. Sun, Q. Deng, and J. Ni. Springer, 2004, pp. 18–25.

[58] K. Huang, X. Jiang, X. Zhang, R. Yan, K. Wang, D. Xiong, and X. Yan. "Energy-Efficient Fault-Tolerant Mapping and Scheduling on Heterogeneous Multiprocessor Real-Time Systems." In: *IEEE Access* 6 (2018), pp. 57614–57630. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2873641.

[59] K.-H. Huang and J. A. Abraham. "Algorithm-Based Fault Tolerance for Matrix Operations." In: *IEEE Trans. Comput.* 33.6 (1984), pp. 518–528.

[60] S. Hwang and C. Kesselman. “Grid workflow: a flexible failure handling framework for the grid.” In: *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*. IEEE. 2003, pp. 126–137.

[61] G. Jacques-Silva, Z. Kalbarczyk, B. Gedik, H. Andrade, K. L. Wu, and R. K. Iyer. “Modeling stream processing applications for dependability evaluation.” In: *Dependable Systems Networks (DSN)*. IEEE, 2011.

[62] R. Jejurikar and R. Gupta. “Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems.” In: *Proceedings of the 2004 international symposium on Low power electronics and design*. ACM. 2004, pp. 78–81.

[63] H. Jin, X.-H. Sun, Z. Zheng, Z. Lan, and B. Xie. “Performance Under Failures of DAG-based Parallel Computing.” In: *CCGRID ’09*. IEEE Computer Society, 2009.

[64] D. Johnson, A. Demers, J. Ullman, M. Garey, and R. Graham. “Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms.” In: *SIAM Journal on Computing* 3.4 (1974), pp. 299–325.

[65] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi. “Characterizing and profiling scientific workflows.” In: *Future Generation Computer Systems* 29.3 (2013), pp. 682–692.

[66] E. Kail, P. fchtpen, and M. Kozlovszky. “A novel adaptive checkpointing method based on information obtained from workflow structure.” In: *Computer Science* 17.3 (2016).

[67] M. C. Kurt, S. Krishnamoorthy, K. Agrawal, and G. Agrawal. “Fault-tolerant Dynamic Task Graph Scheduling.” In: *SC ’14*. IEEE Press, 2014, pp. 719–730.

[68] J.-C. Laprie. “Dependable computing and fault-tolerance.” In: *Digest of Papers FTCS-15* (1985), pp. 2–11.

[69] C.-Y. Lin, T.-H. Chen, and Y.-N. Cheng. “On improving fault tolerance for heterogeneous hadoop mapreduce clusters.” In: *2013 International Conference on Cloud Computing and Big Data*. IEEE. 2013, pp. 38–43.

[70] *List of Green500*. https://www.top500.org/green500/.

[71] C. L. Liu and J. W. Layland. “Scheduling algorithms for multiprogramming in a hard-real-time environment.” In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61.

[72] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra, et al. “Top ten exascale research challenges.” In: *DOE ASCAC subcommittee report* (2014), pp. 1–86.

[73] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[74] R. H. Möhring. “Scheduling under Uncertainty: Bounding the Makespan Distribution.” In: *Computational Discrete Mathematics: Advanced Lectures*. Ed. by H. Alt. Springer, 2001, pp. 79–97.

[75] J. A. Moreno, O. S. Unsal, J. Labarta, and A. Cristal. “NanoCheckpoints: A Task-Based Asynchronous Dataflow Framework for Efficient and Scalable Checkpoint/Restart.” In: *23rd Euromicro PDP*. 2015, pp. 99–102.

[76] S. Moulik, R. Chaudhary, and Z. Das. “HEARS: A heterogeneous energy-aware real-time scheduler.” In: *Microprocessors and Microsystems* 72 (2020), p. 102939. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2019.102939. URL: http://www.sciencedirect.com/science/article/pii/S0141933119302017.

[77] "CHAPTER 1 - Introduction." In: *Architecture Design for Soft Errors*. Ed. by S. Mukherjee. Burlington: Morgan Kaufmann, 2008, pp. 1–41. ISBN: 978-0-12-369529-1. DOI: https://doi.org/10.1016/B978-012369529-1.50003-3. URL: http://www.sciencedirect.com/science/article/pii/B9780123695291500033.

[78] Pegasus. *Pegasus Workflow Generator*. https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator. 2014.

[79] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. 5th. Springer, 2016.

[80] A. Pothen and C. Sun. "A mapping algorithm for parallel sparse Cholesky factorization." In: *SIAM J. on Scientific Computing* 14.5 (1993), pp. 1253–1257.

[81] J. S. Provan and M. O. Ball. "The Complexity of Counting Cuts and of Computing the Probability that a Graph is Connected." In: *SIAM J. Comp.* 12.4 (1983), pp. 777–788.

[82] X. Qin and H. Jiang. "A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems." In: *Parallel Computing* 32.5-6 (2006), pp. 331–356.

[83] W. Qiu, Z. Zheng, X. Wang, and X. Yang. "An efficient fault-tolerant scheduling algorithm for periodic real-time tasks in heterogeneous platforms." In: *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*. June 2013, pp. 1–7. DOI: 10.1109/ISORC.2013.6913213.

[84] *Results with three real frequencies sets*. https://figshare.com/s/7cd9483ae5e56421d0e7.

[85] N. B. Rizvandi, A. Y. Zomaya, Y. C. Lee, A. J. Boloori, and J. Taheri. "Multiple frequency selection in DVFS-enabled processors to minimize energy consumption." In: *Energy-Efficient Distributed Computing Systems* (2012), pp. 443–463.

[86] M. Safari and R. Khorsand. "Energy-aware scheduling algorithm for time-constrained workflow tasks in DVFS-enabled cloud environment." In: *Simulation Modelling Practice and Theory* 87 (2018), pp. 311–326. ISSN: 1569-190X. DOI: https://doi.org/10.1016/j.simpat.2018.07.006. URL: http://www.sciencedirect.com/science/article/pii/S1569190X18300984.

[87] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Vol. 24. Algorithms and Combinatorics. Springer-Verlag, 2003.

[88] D. Sculli. "The Completion Time of PERT Networks." In: *The Journal of the Operational Research Society* 34.2 (1983), pp. 155–158.

[89] S. Shahul and O. Sinnen. "Scheduling Task Graphs Optimally with A*." In: *J. Supercomput.* 51.3 (2010), pp. 310–332.

[90] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. "Fault Tolerant Preconditioned Conjugate Gradient for Sparse Linear System Solution." In: *ICS*. ACM, 2012.

[91] S. Z. Sheikh and M. A. Pasha. "Energy-Efficient Multicore Scheduling for Hard Real-Time Systems: A Survey." In: *ACM Trans. Embed. Comput. Syst.* 17.6 (Dec. 2018), 94:1–94:26. ISSN: 1539-9087. DOI: 10.1145/3291387. URL: http://doi.acm.org/10.1145/3291387.

[92] R. F. da Silva, W. Chen, G. Juve, K. Vahi, and E. Deelman. "Community resources for enabling research in distributed scientific workflows." In: *e-Science (e-Science), 2014 IEEE 10th International Conference on*. Vol. 1. IEEE. 2014, pp. 177–184.

[93] R. M. van Slyke. “Monte Carlo Methods and the PERT Problem.” In: *Operations Research* 11.5 (1963), pp. 839–860.

[94] D. C. Snowdon, G. Van Der Linden, S. M. Petters, and G. Heiser. “Accurate run-time prediction of performance degradation under frequency scaling.” In: *Workshop on Operating Systems Platforms for Embedded Real-Time applications*. 2007, p. 58.

[95] R. Sridharan and R. Mahapatra. “Reliability Aware Power Management for Dual-processor Real-time Embedded Systems.” In: *Proceedings of the 47th Design Automation Conference*. DAC ’10. Anaheim, California: ACM, 2010, pp. 819–824. ISBN: 978-1-4503-0002-5. DOI: 10.1145/1837274.1837480. URL: http://doi.acm.org/10.1145/1837274.1837480.

[96] O. Subasi, G. Yalcin, F. Zyulkyarov, O. Unsal, and J. Labarta. “Designing and Modelling Selective Replication for Fault-Tolerant HPC Applications.” In: *CCGRID*. IEEE, 2017.

[97] O. Subasi, O. S. Ünsal, J. Labarta, G. Yalcin, and A. Cristal. “CRC-Based Memory Reliability for Task-Parallel HPC Applications.” In: *IPDPS*. 2016, pp. 1101–1112.

[98] A. Taherin, M. Salehi, and A. Ejlali. “Reliability-Aware Energy Management in Mixed-Criticality Systems.” In: *IEEE Transactions on Sustainable Computing* (2018).

[99] T. Tobita and H. Kasahara. “A standard task graph set for fair evaluation of multiprocessor scheduling algorithms.” In: *Journal of Scheduling* 5.5 (2002), pp. 379–394.

[100] R. Tolosana-Calasanz, J. Á. Bañares, P. Álvarez, J. Ezpeleta, and O. Rana. “An uncoordinated asynchronous checkpointing model for hierarchical scientific workflows.” In: *Journal of Computer and System Sciences* 76.6 (2010), pp. 403–415.

[101] H. Topcuoglu, S. Hariri, and M.-y. Wu. “Performance-effective and low-complexity task scheduling for heterogeneous computing.” In: *IEEE transactions on parallel and distributed systems* 13.3 (2002), pp. 260–274.

[102] S. Toueg and Ö. Babaoğlu. “On the Optimum Checkpoint Selection Problem.” In: *SIAM J. Comput.* 13.3 (1984).

[103] T. Tsai, Y. Chen, X. He, and C. Li. “STEM: A Thermal-Constrained Real-Time Scheduling for 3D Heterogeneous-ISA Multicore Processors.” In: *IEEE Transactions on Computers* 67.6 (June 2018), pp. 874–889. ISSN: 2326-3814. DOI: 10.1109/TC.2017.2783941.

[104] J. Valdes, R. E. Tarjan, and E. L. Lawler. “The Recognition of Series Parallel Digraphs.” In: *Proc. of STOC’79*. ACM, 1979, pp. 1–12.

[105] E. B. Valentin. “Scheduling hard real-time tasks in heterogeneous multiprocessor platforms subject to energy and temperature constraints.” PhD thesis. Universidade Federal do Amazonas, 2017.

[106] L. G. Valiant. “The Complexity of Enumeration and Reliability Problems.” In: *SIAM J. Comput.* 8.3 (1979), pp. 410–421.

[107] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan. “Replication-Based Fault-Tolerance for Large-Scale Graph Processing.” In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. June 2014, pp. 562–573.

[108] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. “Techniques to reduce the soft error rate of a high-performance microprocessor.” In: *ACM SIGARCH Computer Architecture News* 32.2 (2004), p. 264.

[109] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. "Swift: A language for distributed parallel scripting." In: *Parallel Computing* 37.9 (2011), pp. 633–652.

[110] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, et al. "The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud." In: *Nucleic acids research* (2013), gkt328.

[111] X. Xiao, G. Xie, C. Xu, C. Fan, R. Li, and K. Li. "Maximizing reliability of energy constrained parallel applications on heterogeneous distributed systems." In: *Journal of Computational Science* 26 (2018), pp. 344–353. ISSN: 1877-7503. DOI: https://doi.org/10.1016/j.jocs.2017.05.002. URL: http://www.sciencedirect.com/science/article/pii/S1877750317304933.

[112] G. Xie, G. Zeng, X. Xiao, R. Li, and K. Li. "Energy-Efficient Scheduling Algorithms for Real-Time Parallel Applications on Heterogeneous Distributed Embedded Systems." In: *IEEE Transactions on Parallel and Distributed Systems* 28.12 (Dec. 2017), pp. 3426–3442. ISSN: 2161-9883. DOI: 10.1109/TPDS.2017.2730876.

[113] G. Xie, Y. Chen, X. Xiao, C. Xu, R. Li, and K. Li. "Energy-efficient fault-tolerant scheduling of reliable parallel applications on heterogeneous distributed embedded systems." In: *IEEE Transactions on Sustainable Computing* 3.3 (2018), pp. 167–181.

[114] G. Xie, G. Zeng, R. Li, and K. Li. *Scheduling Parallel Applications on Heterogeneous Distributed Systems*. Springer Singapore, 2019.

[115] H. Xu, R. Li, C. Pan, and K. Li. "Minimizing energy consumption with reliability goal on heterogeneous embedded systems." In: *Journal of Parallel and Distributed Computing* 127 (2019), pp. 44–57.

[116] C. Yang, J. Chen, T. Kuo, and L. Thiele. "An approximation scheme for energy-efficient scheduling of real-time tasks in heterogeneous multiprocessor systems." In: *2009 Design, Automation Test in Europe Conference Exhibition*. Apr. 2009, pp. 694–699. DOI: 10.1109/DATE.2009.5090754.

[117] H.-E. Zahaf, A. E. H. Benyamina, R. Olejnik, and G. Lipari. "Energy-efficient scheduling for moldable real-time tasks on heterogeneous computing platforms." In: *Journal of Systems Architecture* 74 (2017), pp. 46–60. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2017.01.002. URL: http://www.sciencedirect.com/science/article/pii/S138376211730019X.

[118] H.-E. Zahaf, N. Capodieci, R. Cavicchioli, M. Bertogna, and G. Lipari. "A C-DAG task model for scheduling complex real-time tasks on heterogeneous platforms: preemption matters." working paper or preprint. Jan. 2019. URL: https://hal.archives-ouvertes.fr/hal-01971594.

[119] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and H. Abbasi. "Enabling In-situ Execution of Coupled Scientific Workflow on Multi-core Platform." In: *Proc. 26th IEEE IPDPS*. 2012, pp. 1352–1363.

[120] B. Zhao, H. Aydin, and D. Zhu. "Energy management under general task-level reliability constraints." In: *18th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2012, pp. 285–294.

[121] J. Zhou, T. Wei, M. Chen, X. S. Hu, Y. Ma, G. Zhang, and J. Yan. “Variation-aware task allocation and scheduling for improving reliability of real-time MPSoCs.” In: *DATE*. 2018, pp. 171–176.

[122] W. Zhu, H. Chen, and F. Hu. “ASC: Improving spark driver performance with automatic spark checkpoint.” In: *2016 18th International Conference on Advanced Communication Technology (ICACT)*. IEEE. 2016, pp. 607–611.

[123] J. Zhuo and C. Chakrabarti. “System-level energy-efficient dynamic task scheduling.” In: *Proceedings of the 42nd annual Design Automation Conference*. ACM. 2005, pp. 628–631.

List of publications

Articles in International Refereed Journals

[J1] L. Han, L. Canon, H. Casanova, Y. Robert, and F. Vivien. "Checkpointing Workflows for Fail-Stop Errors." In: *IEEE Transactions on Computers* 67.8 (Aug. 2018), pp. 1105–1120. ISSN: 2326-3814. DOI: 10.1109/TC.2018.2801300.

[J2] L. Han, V. L. Fèvre, L.-C. Canon, Y. Robert, and F. Vivien. "A generic approach to scheduling and checkpointing workflows." In: *The International Journal of High Performance Computing Applications* 33.6 (2019), pp. 1255–1274. DOI: 10.1177/1094342019866891. eprint: https://doi.org/10.1177/1094342019866891. URL: https://doi.org/10.1177/1094342019866891.

Articles in International Refereed Conferences

[C1] L. Han, J. Liu, T. Zhou, J. Sun, and X. Chen. "Safety Requirements Specification and Verification for Railway Interlocking Systems." In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. Atlanta, GA, USA, June 2016, pp. 335–340. DOI: 10.1109/COMPSAC.2016.182.

[C2] L. Han, L. Canon, H. Casanova, Y. Robert, and F. Vivien. "Checkpointing Workflows for Fail-Stop Errors." In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. Honolulu, HI, USA, Sept. 2017, pp. 487–497. DOI: 10.1109/CLUSTER.2017.14.

[C3] L. Han, V. Le Fèvre, L.-C. Canon, Y. Robert, and F. Vivien. "A Generic Approach to Scheduling and Checkpointing Workflows." In: *Proceedings of the 47th International Conference on Parallel Processing*. ICPP 2018. Eugene, OR, USA: ACM, 2018, 28:1–28:10. ISBN: 978-1-4503-6510-9. DOI: 10.1145/3225058.3225145. URL: http://doi.acm.org/10.1145/3225058.3225145.

[C4] L. Han, L.-C. Canon, J. Liu, Y. Robert, and F. Vivien. "Improved energy-aware strategies for periodic real-time tasks under reliability constraints." In: *40th IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2019.