Scheduling algorithms and resilience patterns for fail-stop and silent errors

Aurélien Cavelan

Numéro National de Thèse : 2017LYSEN031

THÈSE de DOCTORAT DE L'UNIVERSITE DE LYON

opérée par

l'École Normale Supérieure de Lyon

*École Doctorale N°*512

Informatique et Mathématiques de Lyon

Spécialité : Informatique

présentée et soutenue publiquement le 7 Juillet 2017, par :

Aurélien CAVELAN

Algorithmes d'ordonnancement et schémas de résilience pour les pannes et les erreurs silencieuses

Devant le jury composé de :

Oliver	BEAUMONT	Directeur de recherches, Inria Bordeaux	*Rapporteur*
Anne	BENOIT	Maître de Conférences, ENS de Lyon	*Directrice de thèse*
Luc	GIRAUD	Directeur de recherches, Inria Bordeaux	*Examinateur*
Alain	GIRAULT	Directeur de recherches, Inria Grenoble	*Examinateur*
Amina	GUERMOUCHE	Maître de Conférences, Telecom Paris	*Examinatrice*
Yves	ROBERT	Professeur, ENS de Lyon	*Co-encadrant*
Vaidy	SUNDERAM	Professeur, Emory University, Atlanta, USA	*Rapporteur*

Contents

Introduction

It is one thing to compute a result quickly, but another to get a correct answer. In fact, the reliability of a system is directly proportional to the number of its components, and designing reliable computers is a problem that goes back to the design of Babbage's Analytical Engine itself in 1837 [20]. Frequent errors were common in the first computers: in 1930, the mechanics of the Zuse computers would often get stuck or produce erroneous data [84]. Failures were not unheard of either: in 1950, ENIAC (Electronic Numerical Integrator And Computer), an electronic general-purpose computer, had to replace one of its 17,468 vacuum tubes every two days on average [81]. Altogether, *resilience*, i.e., the capacity to recover quickly from errors or failures, has always been a critical aspect of computer science.

Progress in manufacturing has led to both smaller and more reliable components. However, as the number of processing units increases, the problem remains: Titan, one of the most powerful supercomputers as of 2017 [92], experiences a failure almost every day on average [73]. Meanwhile, simulations can take days or weeks to complete [71], and being able to guarantee the completion and correctness of the computations is crucial for the scientific community. Yet, national agencies all over the world are engaged in a significant research effort to build the first exascale computer [41, 71], a system capable of a billion billion calculations per second. In comparison with current systems, this represents not only a massive increase in computing power, but also in the number of processing units. As a result, the Mean Time Between Failures (MTBF) of the next generation of computers is expected to drop drastically [24, 25, 89].

In February 2014, the ASCAC (Advanced Scientific Computing Advisory Committee) Subcommittee identified resilience as one of the top ten exascale research challenges. The problem is stated as: ensuring correct scientific computation in face of faults, reproducibility, and algorithm verification challenges [71]. This thesis addresses several of these concerns. We first clarify the definition of faults[1], since we consider two different types of faults, fail-stop errors and silent errors.

Fail-stop errors. This phenomenon is well understood. Even if each individual resource is reliable, aggregating too many of them will result in frequent failures globally. For instance, if the MTBF of each individual resource is ten years (a pretty optimistic figure for a processor), then the MTBF of a platform comprising one million of such resources is only five minutes. Specifically, there is a linear correlation between the MTBF of the entire system μ_p, the number

[1] In the rest of this thesis, we use the terms *fault*, *failure* and *error* indifferently.

of resources p and their individual MTBF μ_{ind} [62, Proposition 1.2]:

$$\mu_p = \frac{\mu_{ind}}{p} \,. \tag{1}$$

The standard approach to cope with fail-stop errors is checkpoint, rollback and recovery [28, 46]. Put simply, it allows the application to periodically checkpoint (a.k.a save) the work on a stable storage. Thus, in case of failure, it is possible to rollback the state of the application to the last checkpoint, and restart the execution from there, instead of recomputing everything from scratch. 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 due to errors. One striking result is the formula derived by Young and Daly for the optimal checkpointing period [36, 97]:

$$W^* = \sqrt{2\mu_p C} \,, \tag{2}$$

where C is the checkpoint cost and μ_p is the platform MTBF. This formula is simple and intuitive, and applies to most divisible applications. This corresponds to most iterative kernels, or applications that can be preempted at any time during their execution. However, it does not work for workflow applications, whose tasks are atomic and cannot always be preempted by a checkpoint. The goal is then to find which task to checkpoint, and which task to leave uncheckpointed. For 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 $O(n^3)$ [93]. In this thesis, we will extend these results in Part I and II, respectively.

Silent errors. This phenomenon is not so well understood, but has been recently identified as one of the major challenges for Exascale [25, 74, 76, 89, 102]. There are several causes of silent errors (a.k. a Silent Data Corruptions or SDCs), such as cosmic radiation and packaging pollution, among others. Example of such errors include faults in the Arithmetic Logic Unit (ALU) or bit-flips in the memory. In 2010, Jaguar, which was the most powerful supercomputer at the time [92], was logging errors at a rate of 350 per minute [58]. Even though mechanisms exist to detect such errors at the hardware level, not all errors can be detected [25, 89, 90]. In contrast to a fail-stop error whose detection is immediate, a silent error is identified only when the corrupted data leads to an unusual application behavior. Such a detection latency raises a new challenge: if the error struck before the last checkpoint, and is detected after that checkpoint, then the checkpoint is corrupted and cannot be used for rollback. In order to avoid corrupted checkpoints, an effective approach consists in employing some verification mechanism, and combining it with checkpointing [13, 30, 85]. We study such detectors in Chapter 1. However, while many applications admit fast and accurate detectors such as Algorithm-Based Fault Tolerance (ABFT) [16, 63, 87], thorough and general-purpose error detection can be very costly, and often involves expensive techniques, such as replication [53] or even triplication [72], which are further investigated in Chapter 7.

The rest of the thesis is organized as follows. In Part I, we focus on divisible applications. We start with the study of verification mechanisms in Chapter 1, then we extend Young and Daly's result by combining both fail-stop and silent errors into periodic patterns in Chapter 2,

and we further extend the analysis to multi-level checkpointing with fail-stop errors only in Chapter 3. Part II is dedicated to workflow applications, and the approach is similar to that of Part I. In Chapter 4, we build upon Toueg and Babaoglu's original algorithm to take both types of faults into account. Following these results, we propose several new optimal algorithms and heuristics to tackle the problem of energy consumption in Chapter 5. Finally, we further extend our analysis in Part III. In Chapter 6, we derive the optimal number of processors under different scenarios, and we show the limits of the approach. At last, we use the same approach as in Chapter 7, where we consider replication as a verification and correction mechanism for silent errors. The goal is to identify the optimal replication level based on the number of processors involved. The main contributions of each chapter are summarized below.

Part I

Chapter 1: Coping with recall and precision of soft error detectors. This chapter extends and generalizes two of our previous works [C1, C4]. The main contribution of this work is to characterize the optimal computing pattern for an application: which detector(s) to use, how many detectors of each type to use, together with the length of the work segment that precedes each of them. This work focuses on silent-errors only. We first prove that detectors with imperfect precisions offer limited usefulness. Then we focus on detectors with perfect precision, and we conduct a comprehensive complexity analysis of this optimization problem, showing NP-completeness and designing an FPTAS (Fully Polynomial-Time Approximation Scheme). On the practical side, we provide a greedy algorithm, whose performance is shown to be close to the optimal for a realistic set of evaluation scenarios. Extensive simulations illustrate the usefulness of detectors with false negatives, which are available at a lower cost than the guaranteed detectors. The work in this chapter has been published in the *Journal of Parallel and Distributed Computing* (JPDC) [J1].

Chapter 2: Optimal resilience patterns with fail-stop and silent errors. Based on the results obtained in Chapter 1, as well as in a preliminary analysis [W4, J3], this chapter presents a unified framework and optimal algorithmic solutions addressing both fail-stop and silent errors. Silent errors are handled via verification mechanisms (either partially or fully accurate) and fast in-memory checkpoints. Fail-stop errors are processed via slower disk checkpoints. All verification and checkpoint types are combined into computational patterns. We provide a unified model, and a full characterization of the optimal pattern. Our results nicely extend several published solutions, and demonstrate how to make use of different techniques to solve the double threat of fail-stop and silent errors. Extensive simulations based on real data confirm the accuracy of the model, and show that patterns that combine all resilience mechanisms are required to provide acceptable overheads. The work in this chapter has been published in the proceedings of the *International Parallel & Distributed Processing Symposium* (IPDPS) [C2].

Chapter 3: Towards optimal multi-level checkpointing with fail-stop errors. As opposed to previous chapters, this work focuses on fail-stop errors only. The problem is sim-

ilar to that of Chatpter 2, however dealing with more than two levels of checkpoint makes the analysis much more challenging. We provide a framework to analyze multi-level checkpointing protocols, by formally defining a k-level checkpointing pattern. We provide a first-order approximation to the optimal checkpointing period, and show that the corresponding overhead is in the order of $\sum_{\ell=1}^{k} \sqrt{2\lambda_\ell C_\ell}$, where λ_ℓ is the error rate at level ℓ, and C_ℓ the checkpointing cost at level ℓ. This nicely extends the classical Young/Daly formula on single-level checkpointing. Furthermore, we are able to fully characterize the shape of the optimal pattern (number and positions of checkpoints), and we provide a dynamic programming algorithm to determine the optimal subset of levels to be used. Finally, we perform simulations to check the accuracy of the theoretical study and to confirm the optimality of the subset of levels returned by the dynamic programming algorithm. The results nicely corroborate the theoretical study, and demonstrate the usefulness of multi-level checkpointing with the optimal subset of levels. The work in this chapter has been published in *Transactions on computers* (TC) [J2].

Part II

Chapter 4: Multi-level checkpointing and verification for linear workflows. This chapter focuses on High Performance Computing (HPC) workflows whose dependency graph forms a linear chain. This work extends and generalizes a preliminary analysis [J3], as well as a more recent work [W5]. Similarly to Chapters 2 and 3, we extend single-level checkpointing in two important directions. Our first contribution targets silent errors, and combines in-memory checkpoints with both partial and guaranteed verifications. Our second contribution deals with multi-level checkpointing for fail-stop errors. We present sophisticated dynamic programming algorithms that return the optimal solution for each problem in polynomial time. We also show how to combine all these techniques and solve the general problem with both fail-stop and silent errors. Simulation results demonstrate that these extensions lead to significantly improved performance compared to the standard single-level checkpointing algorithm. The work in this chapter has been published in *Journal of computational science* (JoCS) [J4].

Chapter 5: Voltage overscaling algorithms for energy-efficient workflow computations. In this chapter, we discuss several scheduling algorithms to execute tasks with voltage overscaling. Given a frequency to execute the tasks, operating at a voltage below threshold leads to significant energy savings but also induces timing errors. A verification mechanism must be enforced to detect these errors. As opposed to fail-stop or silent errors, timing errors are deterministic (but unpredictable). For each task, the general strategy is to select a voltage for execution, to check the result, and to select a higher voltage for re-execution if a timing error has occurred, and so on until a correct result is obtained. Switching from one voltage to another incurs a given cost, so it might be efficient to try and execute several tasks at the current voltage before switching to another one. In a preliminary version of this work, we have proposed an optimal polynomial dynamic programming algorithm to solve this problem for a linear chain of tasks [W6]. Determining the optimal solution for independent tasks turns out to be unexpectedly difficult. However, we provide the optimal algorithm for a single task, the optimal algorithm when there are only two voltages, and the optimal level algorithm for a set

of independent tasks, where a level algorithm is defined as an algorithm that executes all remaining tasks when switching to a given voltage. Furthermore, we show that the optimal level algorithm is in fact globally optimal (among all possible algorithms) when voltage switching costs are linear. Finally, we report a comprehensive set of simulations to assess the potential gain of voltage overscaling algorithms. This work has been published in the proceedings of the *Pacific Rim International Symposium on Dependable Computing* (PRDC) [C5].

Part III

Chapter 6: When Amdahl meets Young/Daly. This chapter investigates the optimal number of processors to execute a parallel job, whose speedup profile obeys Amdahl's law, on a large-scale platform subject to fail-stop and silent errors. Without errors, although the speedup is bounded, there is no optimal number of processors: using extra processors will always, even so slightly, benefit the performance. With errors however, adding of processors has the effect of decreasing the platform MTBF (see Equation (1)). We combine the traditional checkpointing and rollback recovery strategies with verification mechanisms to cope with both error sources. We provide an exact formula to express the execution overhead incurred by a periodic checkpointing pattern of length T and with P processors, and we give first-order approximations for the optimal values T^* and P^* as a function of the individual processor failure rate λ_{ind}. A striking result is that P^* is of the order $\lambda_{\text{ind}}^{-1/4}$ if the checkpointing cost grows linearly with the number of processors, and of the order $\lambda_{\text{ind}}^{-1/3}$ if the checkpointing cost stays bounded for any P. We conduct an extensive set of simulations to support the theoretical study. The results confirm the accuracy of first-order approximation under a wide range of parameter settings. This work has been published in the proceedings of *Cluster* [C3].

Chapter 7: Identifying the right replication level for detecting and correcting silent errors. This chapter provides a model and an analytical study of replication as a technique to detect and correct silent errors. Although other detection techniques exist for HPC applications, based on algorithms (ABFT), invariant preservation or data analytics, replication remains the most transparent and least intrusive technique. We explore the right level (duplication, triplication or more) of replication needed to efficiently detect and correct silent errors. Replication is combined with checkpointing and comes with two flavors: *process replication* and *group replication*. Process replication applies to message-passing applications with communicating processes. Each process is replicated, and the platform is composed of process pairs, or triplets. Group replication applies to black-box applications, whose parallel execution is replicated several times. The platform is partitioned into two halves (or three thirds). In both scenarios, results are compared before each checkpoint, which is taken only when both results (duplication) or two out of three results (triplication) coincide. If not, one or more silent errors have been detected, and the application rolls back to the last checkpoint. We provide a detailed analytical study of both scenarios, with formulas to decide, for each scenario, the optimal parameters as a function of the error rate, checkpoint cost, and platform size. We also report a set of extensive simulation results that corroborates the analytical model. This work has been accepted to the *Fault Tolerance for HPC at eXtreme Scale* (FTXS'2017) workshop [W1].

During this thesis, we have addressed a few other problems, and obtained results that are not included in this manuscript. First, this is the case of the work with A. Chien and A. Fang (University of Chicago) on the Global View Resilience (GVR) project, for which a paper has been recently acepted to the *International Conference on Parallel Processing* (ICPP). In this work, we deal with stencil applications and analyze how to limit the scope of re-computations in the recovery phase. We have also published a paper in the proceedings of the *International Workshop on Power-aware Algorithms, Systems, and Architectures* (PASA) [W3]. In this work, we have considered different re-execution speeds: one for the first execution, and another for the re-execution(s) in case of error. We have shown that this approach can help decreasing the overall energy consumption of an application. More recently, we have also derived the optimal checkpointing period for an application replicated on two heterogeneous platforms. This work has been accepted to the *Fault Tolerance for HPC at eXtreme Scale* (FTXS'2017) workshop [W2]. Finally, part of our work on silent errors and verifications has been compiled into a book chapter published in *Emergent Computation* [P1].

Introduction

C'est une chose de calculer un résultat rapidement, mais c'en est une autre d'obtenir une réponse correcte. La fiabilité d'un système est directement proportionnelle au nombre de ses composants, et la conception d'ordinateur fiables est un problème qui remonte à la conception de la Machine Analytique de Babbage elle-même en 1837 [20]. Les erreurs fréquentes étaient communes dans les premiers ordinateurs : en 1930, les mécaniques de l'ordinateur Zuse restaient souvent coincées ou produisaient des résultats erronés [84]. Les pannes n'étaient pas rares non plus : en 1950, ENIAC (Electronic Numerical Integrator And Computer), un ordinateur électronique à usage général, devait remplacer l'un de ses 17568 tubes cathodiques tous les deux jours en moyenne [81]. En fait, la *résilience*, c'est à dire la capacité à récupérer rapidement d'erreurs ou de pannes, a toujours été un aspect critique de l'informatique.

L'amélioration des techniques de fabrication a permis d'obtenir des composants à la fois plus petits et plus fiables. Cependant, comme le nombre d'unités de calculs ne cesse d'augmenter, le problème persiste : Titan, l'un des super-ordinateurs les plus puissants en 2017 [92], est victime d'une panne tous les jours en moyenne [73]. Pendant ce temps, les simulations peuvent mettre des jours ou des semaines avant de terminer [71], et être capable de garantir la terminaison et l'exactitude des calculs est crucial pour la communauté scientifique. Pourtant, les agences nationales tout autour du monde sont engagées dans un effort de recherche important avec pour objectif de construire le premier ordinateur Exascale [41, 71], un système capable d'effectuer un milliard de milliard de calculs par seconde. En comparaison avec les systèmes actuels, cela représente non seulement une très forte augmentation en terme de puissance de calcul, mais aussi en terme de nombre d'unités de calculs. Par conséquent, le temps moyen entre deux fautes, ou Mean Time Between Failure (MTBF) devrait diminuer considérablement [24, 25, 89].

En février 2014, l'ASCAC (Advanced Scientific Computing Advisory Committee) souscomité a identifié la résilience comme l'un des dix défis pour la recherche Exascale. Le problème est défini comme suit : garantir l'exactitude des calculs face aux erreurs, la reproductibilité, et les algorithmes de vérifications [71].

Cette thèse adresse plusieurs de ces problèmes. Nous commençons par clarifier la définition d'erreurs[2], puisque nous considérons deux types d'erreurs, les erreurs de type panne (fail-stop), et les erreurs silencieuses (silent errors).

Pannes. Ce phénomène est bien compris. Même si chaque ressource est fiable individuellement, en agréger trop conduit à des erreurs fréquentes globalement. Par exemple, si le

[2]Dans le reste de cette thèse, nous utilisons les termes*fault*, *failure* et *error* indifféremment.

MTBF de chaque ressource est de 10 ans (un nombre optimiste pour un processeur), alors le MTBF d'une plateforme comptant un million de cette ressource est de cinq minutes seulement. Plus précisément, il y a une corrélation linéaire entre le MTBF du système μ_p, le nombre de ressources p et leur MTBF individuel μ_{ind} [62, Proposition 1.2]:

$$\mu_p = \frac{\mu_{ind}}{p} \, . \tag{1}$$

L'approche standard pour faire face aux pannes est l'utilisation de points de sauvegardes, aussi appelés checkpoints, retours en arrière et récupération [28, 46]. Plus simplement, cela permet à l'application de sauvegarder périodiquement le travail sur un support stable. Ainsi, en cas de panne, il est possible de revenir au dernier checkpoint, et de redémarrer l'application à partir de là, au lieu de tout recalculer depuis de le début. Il y a un compromis évident entre le temps qu'on est prêt à passer à faire des checkpoints, et le temps perdu en ré-exécutions à cause d'erreurs. Un résultat frappant est la formule dérivée par Young et Daly pour la période optimale de checkpoint [36, 97] :

$$W^* = \sqrt{2\mu_p C} \, , \tag{2}$$

où C est le coût de la sauvegarde et μ_p est le MTBF de la plateforme. Cette formule est simple et intuitive, et s'applique à la plupart des applications divisibles. Cela correspond à la plupart des noyaux itératifs ou aux applications pouvant être préemptées par un checkpoint pendant leur exécution. Cependant, cela ne fonctionne par pour les applications type flux de travail, dont les tâches ne peuvent pas être préemptées. L'objectif est alors de trouver quelles tâches sauvegarder, et quelles tâches ne pas sauvegarder. Pour les applications qui peuvent être représentées comme une chaîne de n tâches, Toueg et Babaoglu ont proposé un algorithme à base de programmation dynamique dont la complexité est $O(n^3)$ [93]. Dans cette thèse, nous étendons ces résultats dans la Partie I et II, respectivement.

Erreurs silencieuses. Ce phénomène n'est pas aussi bien compris, mais a été récemment identifié comme l'un des défis majeur pour l'Exascale [25, 74, 76, 89, 102]. Il y a plusieurs sources d'erreurs silencieuses (ou Silent Data Corruptions ou SDCs), comme les radiations cosmiques ou la pollution du au conditionnement, entre autres. Des exemples d'erreurs silencieuses incluent les erreurs dans l'Unité Arithmétique et Logique (UAL) ou des changements de bits dans la mémoire. En 2010, Jaguar, qui étaient l'un des plus puissant super-ordinateur à ce moment [92], enregistrait des erreurs au rythme de 350 par minute [58]. Même si des mécanismes existes pour détecter de telles erreurs au niveau du matériel, toutes les erreurs ne peuvent pas être détectées [25, 89, 90]. Contrairement aux pannes, dont la détection est immédiate, une erreur silencieuse n'est identifiée que lorsque les données corrompues conduisent l'application à se comporter étrangement. Une telle latence de détection soulève un nouveau défi : si une erreur a frappée avant le dernier checkpoint, et n'est détectée qu'après, alors le checkpoint est corrompu et ne peut plus être utilisé pour redémarrer l'application. Afin d'éviter les checkpoints corrompus, une approche efficace consiste à utiliser des mécanismes de vérifications, et à les combiner avec les checkpoints [13, 30, 85]. Nous étudions de tels détecteurs dans le Chapitre 1. Cependant, alors que beaucoup d'applications admettent des vérifications

rapides et précises tel que la technique Algorithm-Based Fault Tolerance (ABFT) [16, 63, 87], une détection exhaustive et générale implique souvent des techniques très couteuses, comme la duplication [53], ou même la triplication [72], qui sont étudiés plus en détails dans le Chapitre 7.

Le reste de la thèse est organisé comme suit. Dans la Partie I, nous nous concentrons sur les applications divisibles. Nous commençons avec l'étude des mécanismes de vérifications dans le Chapitre 1, ensuite nous étendons le résultat des Young et Daly en combinant à la fois les pannes et les erreurs silencieuses dans des schémas de résilience périodique dans le Chapitre 2, et nous étendons cette analyse aux checkpoints multi-niveaux avec pannes seulement dans le Chapitre 3. La Partie II est dédiée aux applications type flux de travail, et suit une approche similaire à celle de la Partie I Dans le Chapitre 4, nous étendons l'algorithme original de Toueg et Babaoglu, puis nous proposons plusieurs nouveaux algorithmes et heuristiques en abordant le problème de la consommation d'énergie dans le Chapitre 5. Finalement, nous poussons un peu plus loin notre analyse dans la Partie III. Dans le Chapitre 6, nous dérivons le nombre optimal de processeurs pour différents scénarios, et nous montrons les limites de l'approche. Enfin, nous utilisons la même approche dans le Chapitre 7, où nous considérons la réplication comme un outils de vérification et de correction pour les erreurs silencieuses. L'objectif est alors de déterminer le niveau de réplication optimal basé sur le nombre de processeurs impliqués. Les contributions principales de chaque chapitre sont résumés ci-dessous.

Partie I

Chapitre 1 : Comment faire face au rappel et la précision des détecteurs derreurs silencieuses. Ce chapitre étends et généralise deux de nos travaux précédents [C1, C4]. La contribution principale de ce travail est de caractériser le schéma de calcul optimal pour une application : combien de détecteurs de chaque type utiliser, ainsi que la longueur du segment de travail qui les précède. Nous prouvons que les détecteurs avec une précision non parfaite sont dune utilité limitée. Ainsi, nous nous concentrons sur des détecteurs avec une précision parfaite et nous menons une analyse de complexité exhaustive de ce problème doptimisation, montrant sa NP-complétude et concevant un schéma FPTAS (Fully Polynomial Time Approximation Scheme). Sur le plan pratique, nous fournissons un algorithme glouton dont la performance est montrée comme étant proche de loptimal pour un ensemble réaliste de scénarios dévaluation. De nombreuses simulations démontrent lutilité de détecteurs avec des résultats faux-négatifs (i.e., des erreurs non détectées), qui sont disponibles à un coût bien moindre que les détecteurs parfaits. Le travail dans ce chapitre a été publié dans le *Journal of Parallel and Distributed Computing* (JPDC) [J1].

Chapitre 2 : Schémas de résilience optimaux pour faire face aux pannes aux erreurs silencieuses. A partir des résultats obtenus dans le Chapitre 1, ainsi qu'une analyse préliminaire [W4, J3], ce chapitre présente un cadre de travail unifié et des solutions algorithmiques optimales pour les pannes et les erreurs silencieuses. Les erreurs silencieuses sont traitées grâce à des mécanismes de vérification (partiellement ou complètement précis) et des checkpoints en mémoire. Des checkpoints sur disques protègent des erreurs fatales. Tous les

types de vérification et checkpoints sont combinés dans un schéma de calcul. Nous donnons un modèle et une caractérisation complète du schéma optimal. Nos résultats étendent de nombreuses solutions déjà publiées, et montrent comment utiliser différentes techniques pour faire face à la double menace des fautes fatales et silencieuses. Des simulations complètes, basées sur des données réelles, confirment la précision du modèle, et montrent que les schémas combinant tous les mécanismes de résilience sont nécessaires pour obtenir des surcoûts acceptables. Le travail dans ce chapitre a été publié dans les proceedings of the *International Parallel & Distributed Processing Symposium* (IPDPS) [C2].

Chapitre 3 : Vers l'optimalité des checkpoints multi-niveaux avec pannes. Contrairement aux chapitres précédents, ce travail se concentre sur les erreurs de type panne (fail-stop) seulement. Le problème est similaire à celui du Chapitre 2, cependant utiliser plusieurs niveaux de checkpoints rend l'analyse beaucoup plus compliquée. Nous caractérisons le schéma optimal, i.e., celui dont le surcoût par unité de calcul est minimal. On montre que ce surcoût minimal est de l'ordre de $\sum_{\ell=1}^{k} \sqrt{2_\ell C_\ell}$, où $_\ell$ est le taux d'erreur au niveau ℓ, et C_ℓ le coût du point de sauvegarde au niveau ℓ. Cette formule étend la célèbre formule de Young et Daly pour un seul niveau. On propose également un algorithme de programmation dynamique pour déterminer le meilleur sous-ensemble de niveaux à utiliser pour minimiser le surcoût global. Enfin, nous conduisons des simulations pour vérifier l'étude théorique, et confirmer l'optimalité du sous-ensemble déterminé par l'algorithme de programmation dynamique. Les résultats corroborent bien l'étude théorique, et montrent toute l'utilité d'une approche multi-niveaux basée sur le sous-ensemble de niveaux optimal. Le travail dans ce chapitre a été publié dans *Transactions on computers* (TC) [J2].

Partie II

Chapitre 4 : Checkpoint multi-niveaux et détection des erreurs silencieuses pour des graphes de tâches linéaires. Ce chapitre se concentre sur des flux de travail, aussi appelés workflows, dont le graphe de dépendance est une chaîne, pour le calcul haute performance. Ce travail étend et généralise une une analyse préliminaire [J3], ainsi qu'un travail plus récent [W5]. Comme pour les Chapitres 2 et 3,nous étendons l'analyse pour les checkpoint à un niveau dans deux directions importantes. Notre première contribution concerne les erreurs silencieuses, et nous combinons les checkpoints avec à la fois des vérifications partielles et garanties. Notre seconde contribution concerne les checkpoints multi-niveaux pour les erreurs de type pannes. Nous présentons un algorithme à base de programmation dynamique sophistiqué qui retourne la solution optimale pour chaque problème en temps polynomial. Nous montrons comment combiner toutes ces techniques pour des applications HPC dont le graphe de dépendances est une chaîne de tâches, et nous donnons plusieurs algorithmes de programmation dynamique qui renvoient la solution optimale en temps polynomial. Des simulations démontrent que lutilisation combinée de checkpoint multi-niveaux et de vérifications améliore la performance. Le travail ans ce chapitre a été publié dans *Journal of computational science* (JoCS) [J4].

Chapitre 5 : Ordonnancement de tâches indépendantes avec réduction drastique du voltage. Dans ce chapitre, nous présentons plusieurs algorithmes dordonnancement pour exécuter des tâches indépendantes avec réduction drastique du voltage. Etant donnée une fréquence pour exécuter les tâches, opérer à un voltage en dessous du seuil limite entraîne des économies dénergie significatives, mais induit également des erreurs de synchronisation. Un mécanisme de vérification doit être utilisé pour détecter ces erreurs. Contrairement aux pannes ou aux erreurs silencieuses, les erreurs de synchronisation sont déterministes (mais imprévisibles). Pour chaque tâche, la stratégie générale consiste à sélectionner un voltage pour lexécution, à vérifier le résultat, à sélectionner un voltage plus élevé pour ré-exécution si une erreur de synchronisation a eu lieu, et ainsi de suite jusquà ce quun résultat correct soit obtenu. Passer dun voltage à un autre a un coût donné, de sorte quil pourrait être efficace dexécuter plusieurs tâches au voltage courant avant den changer. Déterminer la solution optimale se révèle étonnamment difficile. Cependant, nous fournissons lalgorithme optimal pour une seule tâche, lalgorithme optimal lorsquil ny a que deux voltages, et lalgorithme à niveaux optimal pour plusieurs tâches, où un algorithme à niveaux est défini comme étant un algorithme qui exécute toutes les tâches restantes lors du passage à un voltage donné. En outre, nous montrons que lalgorithme à niveaux optimal est en fait globalement optimal (parmi tous les algorithmes possibles) lorsque les coûts de changement de voltage sont linéaires. Enfin, nous présentons un ensemble exhaustif de simulations afin dévaluer le gain potentiel de chacun de nos algorithmes. Ce travail a été publié dans les proceedings of the *Pacific Rim International Symposium on Dependable Computing* (PRDC) [C5].

Partie III

Chapitre 6 : Quand Amdahl rencontre Young et Daly. Ce chapitre étudie le nombre optimal de processeurs pour exécuter un travail parallèle dont le profil d'accélération obéit à la loi d'Amdahl, sur une plateforme à grande échelle exposée aux pannes et aux erreurs silencieuses. Nous combinons l'approche traditionnelle de checkpointing/recovery avec des mécanismes de vérification pour faire face aux deux types d'erreurs. Nous fournissons une formule exacte pour mesurer le surcoût du temps d'exécution induit par un motif de checkpoint périodique de longueur T et avec P processeurs, et nous donnons une approximation au premier ordre des valeurs optimales de T^* et P^* en fonction du taux d'erreur individuel d'un processeur . Un résultat frappant est que P^* est de l'ordre de $^{-1/4}$ quand le coût de checkpoint croît linéairement avec le nombre de processeurs, et de l'ordre de $^{-1/3}$ quand le coût de checkpoint reste borné par P. Nous menons une large campagne de simulations pour appuyer l'étude théorique. Les résultats confirmes la précision de l'approximation au premier ordre pour une large gamme de paramètres. Ce travail a été publié dans les proceedings of *Cluster* [C3].

Chapitre 7 : Quel est le bon niveau de réplication pour détecter et corriger les erreurs silencieuses?. Ce chapitre propose un modèle et une étude analytique de la réplication en tant que technique pour détecter et corriger les erreurs silencieuses. Bien que d'autres techniques existent pour les applications HPC, basées sur des algorithmes (ABFT), préservation d'invariant, ou analyse de données, la réplication reste la technique la plus transparente

et la moins intrusive. Nous explorons le bon niveau (duplication, triplication ou plus) de réplication nécessaire pour détecter et corriger les erreurs silencieuses de manière efficace. La réplication est combinée avec des checkpoints et se présente sous deux formes : *réplication de processus* et *réplication de groupes*. La réplication de processus s'applique aux applications à passage de messages avec des processus communicants. Chaque processus est répliqué, et la plate-forme est composée de paires, ou triplets de processus. La réplication de groupe s'applique à des applications type boîte noire, dont l'exécution parallèle est répliquée plusieurs fois. La plate-forme est alors partitionnée en deux moitiés (ou trois tiers). Dans les deux scénarios, les résultats sont comparés avant chaque checkpoint, qui est effectué seulement lorsque les deux résultats (duplication) ou deux sur trois (triplication) coïncident. Sinon, une ou plusieurs erreurs silencieuses ont été détectées, et l'application redémarre depuis le dernier checkpoint. Nous proposons une étude analytique détaillée des deux scénarios ainsi que les paramètres optimaux fonction du taux d'erreur, du coût du checkpoint, et de la taille de la plate-forme. Nous donnons également les résultats d'un ensemble de simulations qui viennent corroborer le modèle analytique. Ce travail a été accepté au *Fault Tolerance for HPC at eXtreme Scale* (FTXS'2017) workshop [W1].

Durant cette thèse, nous avons adressé également adressé d'autres problèmes qui ne sont pas inclues dans ce manuscrit. Tout d'abord, c'est ke cas du travail réalisé avec A. Chien et A. Fang (Université de Chicago) sur le projet Global View Resilience (GVR), pour lequel un papier a récemment été accepté à *International Conference on Parallel Processing* (ICPP). Dans ce travail, nous considérons des applications de type stencil et nous analysons comment limiter le nombre d'opérations à recalculer en cas d'erreur durant le phase de récupération. Nous avons également publié un papier dans les proceedings of the *International Workshop on Power-aware Algorithms, Systems, and Architectures* (PASA) [W3]. Dans ce travail, nous avons considéré différentes vitesses de ré-exécution : une pour la première exécution, et une autre pour les ré-exécution en cas d'erreurs. Nous avons montré que cette approche aide à réduire la consommation d'énergie globale de l'application. Plus récemment, nous avons également dérivé la période optimale de checkpoint pour une application répliquée sur deux machines hétérogènes. Ce travail a été accepté au *Fault Tolerance for HPC at eXtreme Scale* (FTXS'2017) workshop [W2]. Enfin, une partie de notre travail sur les erreurs silencieuse et les vérifications a été publié dans un chapitre de livre dans *Emergent Computation* [P1].

Part I

Resilience Patterns

Chapter 1

Coping with Recall and Precision of Soft Error Detectors

This chapter extends and generalizes two of our previous works [C1, C4]. The main contribution of this work is to characterize the optimal computing pattern for an application: which detector(s) to use, how many detectors of each type to use, together with the length of the work segment that precedes each of them. This work focuses on silent-errors only. We first prove that detectors with imperfect precisions offer limited usefulness. Then we focus on detectors with perfect precision, and we conduct a comprehensive complexity analysis of this optimization problem, showing NP-completeness and designing an FPTAS (Fully Polynomial-Time Approximation Scheme). On the practical side, we provide a greedy algorithm, whose performance is shown to be close to the optimal for a realistic set of evaluation scenarios. Extensive simulations illustrate the usefulness of detectors with false negatives, which are available at a lower cost than the guaranteed detectors. The work in this chapter has been published in the *Journal of Parallel and Distributed Computing* (JPDC) [J1].

1.1 Introduction

In order to avoid corrupted checkpoints, an effective approach consists in employing some verification mechanism and combining it with checkpointing [3, 13, 30, 85]. This verification mechanism can be general-purpose (e.g., based on replication [53] or even triplication [72]) or application-specific (e.g., based on Algorithm-based fault tolerance (ABFT) [16, 63, 87], on approximate re-execution for ODE and PDE solvers [14], or on orthogonality checks for Krylov-based sparse solvers [30, 85]).

The simplest protocol with this approach would be to execute a verification procedure before taking each checkpoint. If the verification succeeds, then one can safely store the checkpoint. Otherwise, it means that an error has struck since the last checkpoint, which was duly verified, and one can safely recover from that checkpoint to resume the execution of the application. Of course, more sophisticated protocols can be designed, by coupling multiple verifications with one checkpoint, or interleaving multiple checkpoints and verifications [3, 13]. The optimal parameter (e.g., number of verifications per checkpoint) in these protocols would be

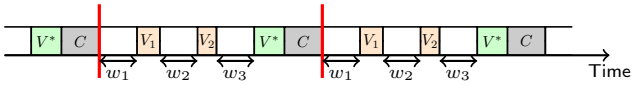


Figure 1.1: A periodic pattern (highlighted in red) with three segments, two partial verifications and a verified checkpoint.

determined by the relative cost of executing a verification.

In practice, not all verification mechanisms are 100% accurate and at the same time admit fast implementations. In fact, guaranteeing accurate and efficient detection of silent errors for scientific applications is one of the hardest challenges towards extreme-scale computing [24, 25]. Indeed, thorough and general-purpose error detection is usually very costly, and often involves expensive techniques, such as replication [53] or even triplication [72], which are further investigated in Chapter 7. Many applications have developed specific verification mechanisms that leverage detailed knowledge of the physics behind the simulation to determine whether the output of a simulation is corruption-free or not. While such application-specific mechanisms do not detect the totality of SDC affecting the hardware, they can guarantee to detect all the corruptions relevant for the end user, thus they can be called arguably *perfect detectors* or *guaranteed verifications*, at least from the user's perspective. For many parallel applications, alternative techniques exist that are capable of detecting silent errors but with lower accuracy. We call these techniques *partial verifications*. One example is the lightweight SDC detector based on data dynamic monitoring [9], designed to recognize anomalies in HPC datasets based on physical laws and spatial interpolation. Similar fault filters have also been designed to detect silent errors based on time series predictions [15]. Although not completely accurate, these partial verification techniques nevertheless cover a substantial number of silent errors, and more importantly, they incur very low overheads. These properties make them attractive candidates for designing more efficient resilient protocols.

Since checkpointing is often expensive in terms of both time and space required, to avoid saving corrupted data, we only keep *verified checkpoints* by placing a guaranteed verification right before each checkpoint. Such a combination ensures that the checkpoint contains valid data and can be safely written onto stable storage. The execution of the application is partitioned into *periodic patterns*, i.e., computational chunks that repeat over time, and that are delimited by verified checkpoints, possibly with a sequence of partial verifications in between. Figure 1.1 shows a periodic pattern with two partial verifications followed by a verified checkpoint.

The error detection accuracy of a partial verification can be characterized by two parameters: recall and precision. The *recall*, denoted by r, is the ratio between the number of detected errors and the total number of errors that occurred during a computation. The *precision*, denoted by p, is the ratio between the number of true errors and the total number of errors detected by the verification. For example, a basic spatial based SDC detector [9] has been shown to have a recall value around 0.5 and a precision value very close to 1, which means that it is capable of detecting half of the errors with almost no false alarm. A guaranteed verification can be considered as a special type of partial verification with recall $r^* = 1$ and precision $p^* = 1$. Each partial verification also has an associated *cost* V, which is typically much smaller

than the cost V^* of a guaranteed verification. Note that precision and recall are conflicting objectives as they both are directly related to the allowed prediction error of the detector. If the prediction error is too small, then small changes in data behavior will produce false positives. On the other hand, if the allowed prediction error is too large, important corruption could be absorbed in the error corrupting the execution. Thus, one usually sets a target for one of them (e.g., precision $= 0.999$) and then measures the recall obtained with such a level of precision. Therefore, although it is hard to know in advance the precision and recall of a given detector for a particular application, it is possible to set a target for either one, and then quickly measure the complementary parameter.

An application can use several types of detectors with different overheads and accuracies. For instance, to detect silent errors in HPC datasets, one has the option of using either a detector based on time series prediction [15], or a detector using spatial multivariate interpolation [9]. The first one needs more data to make a prediction, hence comes at a higher cost. However, its accuracy is also better. In the example of Figure 1.1, the second verification may use a detector whose cost is lower than that of the first one, i.e., $V_2 < V_1$, but is expected to have a lower accuracy as well, i.e., $r_2 < r_1$ and/or $p_2 < p_1$. This is due to the fact that less accurate detectors perform a much simpler approximation, leading to more prediction errors.

In this chapter, we assume that we have several detector types, whose costs and accuracies may differ. At the end of each segment inside the pattern, any detector can be used. The only constraint is to enforce a guaranteed verification after the last segment. Given the values of C (cost to checkpoint) and V^* (cost of guaranteed verification), as well as the cost $V^{(j)}$, recall $r^{(j)}$ and precision $p^{(j)}$ of each detector type $D^{(j)}$, the main question is which detector(s) to use? Note that we do not assume that all detectors perform equally on all applications, nor that their efficiency can be easily predicted for each type of application. The only requirement is that the accuracy and cost of those detectors can be measured in a relatively easy way. The objective is to find the optimal pattern that minimizes the expected execution time of the application. Intuitively, including more partial verifications in a pattern allows us to detect more errors earlier in the execution, thereby reducing the waste due to re-execution; but that comes at the price of additional overhead in an error-free execution, and in case of bad precision, of unnecessary rollbacks and recoveries. Therefore, an optimal strategy must seek a good tradeoff between error-induced waste and error-free overhead. The problem is intrinsically combinatorial, because there are many parameters to choose: the length of the pattern, the number of partial verifications, and the type and location of each partial verification within the pattern. Of course, the length of an optimal pattern will also depend on the platform MTBF μ.

When there is a single segment in the pattern without intermediate verification, the only thing to determine is the size of the segment. In the classical protocol for fail-stop errors (where verification is not needed), the optimal checkpointing period is known to be $\sqrt{2\mu C}$ (where C is the checkpoint time), as given by Young [97] and Daly [36]. This formula provide first-order approximation to the length of the optimal pattern in the corresponding scenario, and is valid only if $C \ll \mu$. While most applications accept several detector types, there has been no attempt to determine which and how many of these detectors should be used. This work is the first to investigate the use of different types of partial detectors while taking both recall and precision into consideration.

As in those previous works, we apply first-order approximation to tackle the optimization

problem. We first show that a partial detector with imperfect precision plays a limited role in the optimization of a pattern. Then we focus on detectors with perfect precision but imperfect recall, and we prove that the optimization problem is NP-complete. In this case, a detector is most useful when it offers the highest *accuracy-to-cost ratio*, defined as $\phi^{(j)} = \frac{a^{(j)}}{b^{(j)}}$, where $a^{(j)} = \frac{r^{(j)}}{2-r^{(j)}}$ denotes the accuracy of the detector and $b^{(j)} = \frac{V^{(j)}}{V^*+C}$ the relative cost. Finally, we propose a greedy algorithm and a fully polynomial-time approximation scheme (FPTAS) to solve the problem. Simulation results, based on a wide range of parameters from realistic detectors, corroborate the theoretical study by showing that the detector with the best accuracy-to-cost ratio should be favored. In some particular cases with close accuracy-to-cost ratios, an optimal pattern may use multiple detector types, but the greedy algorithm has been shown to work really well in these scenarios.

The rest of this chapter is organized as follows. Section 1.2 surveys the related work. Section 1.3 introduces the model, notations and assumptions. Section 1.4 computes the expected execution time of a given pattern, based on which we derive some key properties of the optimal pattern in Section 1.5. Section 1.6 provides a comprehensive complexity analysis. While the optimization problem is shown to be NP-complete, a simple greedy algorithm is presented, and a fully polynomial-time approximation scheme is described. Simulation results are presented in Section 1.7. Finally, Section 1.8 provides concluding remarks.

1.2 Related work

Considerable efforts have been directed at detection techniques to reveal silent errors. Hardware mechanisms, such as ECC memory, can detect and even correct a fraction of errors. Unfortunately, future extreme scale systems are expected to observe an important increase in soft errors, aggravated by power constraints at increased system size. Most traditional resilient approaches maintain a single checkpoint. If the checkpoint file contains corrupted data, the application faces an irrecoverable failure and must restart from scratch. This is because error detection latency is ignored in traditional rollback and recovery schemes, which assume instantaneous error detection (therefore mainly targeting fail-stop errors) and are unable to accommodate SDC. This section describes some related work on detecting and handling silent errors.

1.2.1 Checkpoint versioning

One approach to dealing with silent errors is by maintaining several checkpoints in memory [70]. This multiple-checkpoint approach, however, has three major drawbacks. First, it is very demanding in terms of stable storage: each checkpoint typically represents a copy of a large portion of the memory footprint of the application, which may well correspond to tens or even hundreds of terabytes. Second, the application cannot be recovered from fatal failures: suppose we keep k checkpoints in memory, and a silent error has struck before all of them. Then, all live checkpoints are corrupted, and one would have to re-execute the entire application from scratch. Third, even without memory constraints, we have to determine which checkpoint is the last valid one, which is needed to safely recover the application. However,

due to the detection latency, we do not know when the silent error has occurred, hence we cannot identify the last valid checkpoint.

1.2.2 Process replication

There are few methods that can guarantee a perfect detection recall. Process replication is one of them. The simplest technique is triple modular redundancy and voting [72]. Elliot et al. [45] propose combining partial redundancy and checkpointing, and confirm the benefit of dual and triple redundancy. Fiala et al. [53] apply process replication (each process is equipped with a replica, and messages are quadruplicated) in the RedMPI library for high-performance scientific applications. Ni et al. [75] use checkpointing and replication to detect and enable fast recovery of applications from both silent errors and hard errors. Chapter 7 considers replication as a detection and correction mechanism. However, full process replication is generally too expensive to be used in extreme scale HPC systems and is usually avoided for this reason.

1.2.3 Application-specific techniques

Application-specific information can be very useful to enable ad-hoc solutions, which dramatically decrease the cost of detection. Algorithm-based fault tolerance (ABFT) [16, 63, 87] is a well-known technique, which uses checksums to detect up to a certain number of errors in linear algebra kernels. Unfortunately, ABFT can only protect datasets in linear algebra kernels, and it must be implemented for each different kernel, which incurs a large amount of work for large HPC applications. Other techniques have also been advocated. Benson, Schmit and Schreiber [14] compare the result of a higher-order scheme with that of a lower-order one to detect errors in the numerical analysis of ODEs and PDEs. Sao and Vuduc [85] investigate self-stabilizing corrections after error detection in the conjugate gradient method. Bridges et al. [19] propose linear solvers to tolerant soft faults using selective reliability. Elliot et al. [44] design a fault-tolerant GMRES capable of converging despite silent errors. Bronevetsky and de Supinski [21] provide a comparative study of detection costs for iterative methods.

1.2.4 Analytics-based corruption detection

Recently, several SDC detectors based on data analytics have been proposed, showing promising results. These detectors use several interpolation techniques such as time series prediction [15] and spatial multivariate interpolation [8, 9, 11]. Such techniques have the benefit of offering large detection coverage for a negligible overhead. However, these detectors do not guarantee full coverage; they can detect only a certain percentage of corruptions (i.e., partial verification with an imperfect recall). Nonetheless, the accuracy-to-cost ratios of these detectors are high, which makes them interesting alternatives at large scale. Similar detectors have also been designed to detect SDCs in the temperature data of the Orbital Thermal Imaging Spectrometer (OTIS) [31]. Most of the research work done in this domain focuses on how to increase the error detection accuracy while keeping low overhead, but there has been no theoretical attempt to find the optimal protocol the applications should use when multiple verification techniques are offered by the runtime.

1.2.5 Optimal strategies with guaranteed verifications

Theoretically, various protocols that couple verification and checkpointing have been studied. Aupy et al. [3] propose and analyze two simple patterns: one with k checkpoints and one verification, and the other with k verifications and one checkpoint, which needs to maintain only one checkpoint. Benoit et al. [13] extend the analysis of [3] by including p checkpoints and q verifications that are interleaved to form arbitrary patterns. All of these results assume the use of guaranteed verifications only.

The only analysis that includes partial verifications in the pattern is the preliminary versions of this work [C1, C4]. However, [C4] restricts to a single type of partial verification, and both [C4] and [C1] focuses on verifications with perfect precision. In this chapter, we provide the first theoretical analysis that includes partial verifications of different types, and that considers verifications with imperfect precision.

1.3 Model

We consider divisible-load applications, where checkpoints and verifications can be inserted anywhere in the execution of the application. The occurrence of silent errors follows a Poisson process with arrival rate $\lambda = \frac{1}{\mu}$, where μ denotes the MTBF of the platform.

We enforce resilience through the use of a *pattern* that repeats periodically throughout the execution, as discussed in Section 1.1. When an error alarm is raised inside the pattern, either by a partial verification or by the guaranteed one, we roll back to the beginning of the pattern and recover from the last checkpoint (taken at the end of the execution of the previous pattern, or initial data for the first pattern). Since the last verification of the pattern is guaranteed, we need to maintain only one checkpoint at any time, and it is always valid. The objective is to find a pattern that minimizes the expected execution time of the application.

Let C denote the cost of checkpointing, R the cost of recovery and V^* the cost of guaranteed verification. Furthermore, there are k types of detectors available, and the detector type $D^{(j)}$, where $1 \leq j \leq k$, is characterized by its cost $V^{(j)}$, recall $r^{(j)}$ and precision $p^{(j)}$. For notational convenience, we also define $g^{(j)} = 1 - r^{(j)}$ (proportion of undetected errors) and let D^* be the guaranteed detector with cost V^*, recall $r^* = 1$ and precision $p^* = 1$.

A pattern $\text{PATTERN}(W, n, \boldsymbol{\alpha}, \mathbf{D})$ is defined by its total length W, its total number n of segments, a vector $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \ldots, \alpha_n]^T$ containing the proportions of the segment sizes, and a vector $\mathbf{D} = [D_1, D_2, \ldots, D_{n-1}, D^*]^T$ containing the detectors used at the end of each segment. We also define the vector of segment sizes $\mathbf{w} = [w_1, w_2, \ldots, w_n]^T$. Formally, for each segment i, where $1 \leq i \leq n$, w_i is the size of the segment, $\alpha_i = \frac{w_i}{W}$ is the proportion of the segment size in the whole pattern, and D_i is the detector used at the end of the segment. We have $\sum_{i=1}^{n} \alpha_i = 1$, and $\sum_{i=1}^{n} w_i = W$. If $i < n$, D_i has cost V_i, recall r_i and precision p_i (we have $D_i = D^{(j)}$ for some type j, $1 \leq j \leq k$), and $D_n = D^*$ with cost V^*, recall $r^* = 1$ and precision $p^* = 1$. Note that the same detector type $D^{(j)}$ may well be used at the end of several segments. Let $g_i = 1 - r_i$ denote the probability that the i-th detector of the pattern fails to detect an error (for $1 \leq i < n$), and let $g_{[i,j[} = \prod_{k=i}^{j-1} g_k$ be the probability that the error remains undetected by detectors D_i to D_{j-1} (for $1 \leq i < j < n$). Similarly, p_i represents

the probability that the i-th detector does not raise a false alarm when there is no error, and let $p_{[i,j[} = \prod_{k=i}^{j-1} p_k$ denote the probability that no false alarm is raised by detectors D_i to D_{j-1}. In the example of Figure 1.1, we have $W = w_1 + w_2 + w_3$ and $n = 3$. The first partial verification has cost V_1, recall r_1 and precision p_1, and the second one has cost V_2, recall r_2 and precision p_2.

Let W_{base} denote the base time of an application without any overhead due to resilience techniques (without loss of generality, we assume unit-speed execution). Suppose the execution is divided into periodic patterns, defined by $\text{PATTERN}(W, n, \boldsymbol{\alpha}, \mathbf{D})$. Let $\mathbb{E}(W)$ be the expected execution time of the pattern. Then, the expected makespan W_{final} of the application when taking silent errors into account can be bounded as follows:

$$\left\lfloor \frac{W_{\text{base}}}{W} \right\rfloor \times \mathbb{E}(W) \leq W_{\text{final}} \leq \left\lceil \frac{W_{\text{base}}}{W} \right\rceil \times \mathbb{E}(W).$$

This is because the execution involves $\left\lfloor \frac{W_{\text{base}}}{W} \right\rfloor$ full patterns, and terminates by a (possibly) incomplete one. For large jobs, we can approximate the execution time as

$$W_{\text{final}} \approx \frac{\mathbb{E}(W)}{W} \times W_{\text{base}}.$$

Let $H(W) = \frac{\mathbb{E}(W)}{W} - 1$ denote the execution *overhead* of the pattern. We obtain $W_{\text{final}} \approx W_{\text{base}} + H(W) \times W_{\text{base}}$. Thus, minimizing the expected makespan is equivalent to minimizing the pattern overhead $H(W)$.

We assume that errors only strike the computations, while verifications and I/O transfers (checkpointing and recovery) are protected and are thus error-free. It is shown in Chapter 2 that removing this assumption does not affect the asymptotic behavior of a pattern. We also assume statistical independence of the detectors: if the same detector is applied twice, say at time steps t_1 and t_2, then its recall and precision are the same for both instances. This is because detectors actually detect the effect of an error on the resulting data, rather than the error itself. When an error is missed the first time at step t_1, either it dissipates and becomes harmless, or it propagates and corrupts more data. In the latter case, running the detector again after some iterations at step t_2 will actually detect the error within the precision and recall of the detector. Furthermore, to be on the safe side, we never use two detectors in a row. The idea is that after a chunk of computations, output data will be considered as random input when fed to the next detector. Understanding error propagation and correlation requires deep knowledge of the application. In this work, we provide a general-purpose solution, hence we have to rely on the independence hypothesis.

1.4 Expected execution time of a pattern

In this section, we compute the expected execution time of a pattern by giving a closed-form formula that is exact up to second-order terms. This is a key result that will be used to derive properties of the optimal pattern in the subsequent analysis.

Consider a given pattern $\text{PATTERN}(W, n, \boldsymbol{\alpha}, \mathbf{D})$. The following proposition shows the expected execution time of this pattern.

Proposition 1. *The expected time to execute a pattern* PATTERN$(W, n, \boldsymbol{\alpha}, \mathbf{D})$ *is*

$$\mathbb{E}(W) = \sum_{i=1}^{n} \frac{W\alpha_i + V_i}{p_{[i,n[}} + C + \left(\frac{1}{p_{[1,n[}} - 1\right) R + \lambda W \left(\frac{R}{p_{[1,n[}} + W\boldsymbol{\alpha}^T M \boldsymbol{\alpha} + \boldsymbol{\alpha}^T M \mathbf{v}\right) + o(\lambda), \tag{1.1}$$

where $\mathbf{v}$ *is an* $n \times 1$ *vector defined by* $\mathbf{v} = [V_1, V_2, \ldots, V_n]^T$*, and* M *is an* $n \times n$ *matrix defined as* $M_{ij} = \frac{1}{p_{[i,n[}}$ *for* $i \leq j$ *and* $M_{ij} = \frac{g_{[i,j[}}{p_{[j,n[}}$ *for* $i > j$*.*

Proof. Let q_i denote the probability that an error occurs in the execution of segment i. We can express the expected execution time of the pattern recursively as follows:

$$\begin{aligned}\mathbb{E}(W) = &\left(\prod_{k=1}^{n}(1-q_k)\right) p_{[1,n[}C + \left(1 - \left(\prod_{k=1}^{n}(1-q_k)\right) p_{[1,n[}\right)(R + \mathbb{E}(W)) \\ &+ \sum_{i=1}^{n}\left(\sum_{j=1}^{i-1}\left(\prod_{k=1}^{j-1}(1-q_k)\right) p_{[1,j[}q_j g_{[j,i[} + \left(\prod_{k=1}^{i-1}(1-q_k)\right) p_{[1,i[}\right)(w_i + V_i).\end{aligned} \tag{1.2}$$

The first line shows that checkpointing will be taken only if no error has occurred in all the segments and no intermediate detector has raised a false alarm. This happens with probability

$$\left(\prod_{k=1}^{n}(1-q_k)\right)\left(\prod_{k=1}^{n-1} p_k\right) = \left(\prod_{k=1}^{n}(1-q_k)\right) p_{[1,n[}. \tag{1.3}$$

In all the other cases, the application needs to recover from the last checkpoint and then recomputes the entire pattern. The second line shows the expected cost involved in the execution of each segment of the pattern and the associated verification. To better understand it, let us consider the third segment of size w_3 and the verification D_3 right after it, which will be executed only when the following events happen (with the probability of each event in brackets):

- There is a fault in the first segment (q_1), which is missed by the first verification ($1 - r_1 = g_1$) and again missed by the second verification ($1 - r_2 = g_2$).
- There is no fault in the first segment ($1 - q_1$), the first verification does not raise a false alarm (p_1), and there is a fault in the second segment (q_2), which is missed by the second verification ($1 - r_2 = g_2$).
- There is no fault in the first segment ($1 - q_1$), the first verification does not raise a false alarm (p_1), there is no fault in the second segment ($1 - q_2$), and the second verification does not raise a false alarm (p_2).

Thus, the expected cost involved in the execution of this segment is given by

$$\begin{aligned}&\Big(q_1 g_1 g_2 + (1-q_1)p_1 q_2 g_2 + (1-q_1)p_1(1-q_2)p_2\Big)(w_3 + V_3) \\ &= \Big(q_1 g_{[1,3[} + (1-q_1)p_{[1,2[}q_2 g_{[2,3[} + (1-q_1)(1-q_2)p_{[1,3[}\Big)(w_3 + V_3).\end{aligned}$$

We can generalize this reasoning to express the expected cost to execute the i-th segment of the pattern, which leads to Equation (1.2).

Since errors arrive according to the Poisson process, by definition, we have $q_i = 1 - e^{-\lambda w_i}$. Substituting it into the recursive formula and solving for $\mathbb{E}(W)$, we obtain the expected execution time as

$$\mathbb{E}(W) = C + \left(\frac{e^{\lambda W}}{p_{[1,n[}} - 1\right) R + \sum_{i=1}^{n} \left(\sum_{j=1}^{i-1} \left(e^{\lambda W_{j,n}} - e^{\lambda W_{j+1,n}}\right) \frac{g_{[j,i[}}{p_{[j,n[}} + \frac{e^{\lambda W_{i,n}}}{p_{[i,n[}} \right) (w_i + V_i),$$

where $W_{i,j} = \sum_{k=i}^{j} w_k$. Approximating $e^{\lambda x} = 1 + \lambda x + o(\lambda)$ up to the first-order term, we can further simplify the expected execution time as

$$\begin{aligned}
\mathbb{E}(W) &= C + \left(\frac{1 + \lambda W}{p_{[1,n[}} - 1\right) R + \sum_{i=1}^{n} \left(\sum_{j=1}^{i-1} \frac{\lambda w_j g_{[j,i[}}{p_{[j,n[}} + \frac{1 + \lambda \sum_{j=i}^{n} w_j}{p_{[i,n[}} \right) (w_i + V_i) + o(\lambda) \\
&= \sum_{i=1}^{n} \frac{w_i + V_i}{p_{[i,n[}} + C + \left(\frac{1}{p_{[1,n[}} - 1\right) R \\
&\qquad + \lambda W \frac{R}{p_{[1,n[}} + \lambda \sum_{i=1}^{n} \left(\sum_{j=1}^{i-1} \frac{w_j g_{[j,i[}}{p_{[j,n[}} + \sum_{j=i}^{n} \frac{w_j}{p_{[i,n[}} \right) (w_i + V_i) + o(\lambda).
\end{aligned}$$

Letting $F = \sum_{i=1}^{n} \left(\sum_{j=1}^{i-1} \frac{w_j g_{[j,i[}}{p_{[j,n[}} + \sum_{j=i}^{n} \frac{w_j}{p_{[i,n[}} \right) (w_i + V_i)$, we can express it in the following matrix form:

$$F = \mathbf{w}^T M \mathbf{w} + \mathbf{w}^T M \mathbf{v},$$

where M is the following $n \times n$ matrix:

$$M = \begin{bmatrix}
\frac{1}{p_{[1,n[}} & \frac{1}{p_{[1,n[}} & \frac{1}{p_{[1,n[}} & \cdots & \frac{1}{p_{[1,n[} \\
\frac{g_{[1,2[}}{p_{[1,n[}} & \frac{1}{p_{[2,n[}} & \frac{1}{p_{[2,n[}} & \cdots & \frac{1}{p_{[2,n[} \\
\frac{g_{[1,3[}}{p_{[1,n[}} & \frac{g_{[2,3[}}{p_{[2,n[}} & \frac{1}{p_{[3,n[}} & \cdots & \frac{1}{p_{[3,n[} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\frac{g_{[1,n[}}{p_{[1,n[}} & \frac{g_{[2,n[}}{p_{[2,n[}} & \frac{g_{[3,n[}}{p_{[3,n[}} & \cdots & \frac{1}{p_{[n,n[}}
\end{bmatrix}.$$

For instance, when $n = 4$, we have:

$$M = \begin{bmatrix}
\frac{1}{p_1 p_2 p_3} & \frac{1}{p_1 p_2 p_3} & \frac{1}{p_1 p_2 p_3} & \frac{1}{p_1 p_2 p_3} \\
\frac{g_1}{p_1 p_2 p_3} & \frac{1}{p_2 p_3} & \frac{1}{p_2 p_3} & \frac{1}{p_2 p_3} \\
\frac{g_1 g_2}{p_1 p_2 p_3} & \frac{g_2}{p_2 p_3} & \frac{1}{p_3} & \frac{1}{p_3} \\
\frac{g_1 g_2 g_3}{p_1 p_2 p_3} & \frac{g_2 g_3}{p_2 p_3} & \frac{g_3}{p_3} & 1
\end{bmatrix}.$$

Now, by using $\mathbf{w} = W\boldsymbol{\alpha}$, we obtain Equation (1.1), which completes the proof of the proposition. □

1.5 Properties of optimal pattern

In this section, we characterize the properties of the optimal pattern. First, we derive the optimal length of a pattern (Section 1.5.1). Then, we show that the optimal pattern does not contain partial detectors with imperfect precision (Section 1.5.2). By focusing on detectors with perfect precision, we define two key parameters to characterize a pattern (Section 1.5.3). Finally, we obtain the optimal positions for a give set of partial verifications (Section 1.5.4).

1.5.1 Optimal length of a pattern

We first compute the optimal length W of a pattern $\text{PATTERN}(W, n, \boldsymbol{\alpha}, \mathbf{D})$ in order to minimize its execution overhead $H(W)$.

Theorem 1. *The execution overhead of a pattern* $\text{PATTERN}(W, n, \boldsymbol{\alpha}, \mathbf{D})$ *is minimized when its length is*

$$W^* = \sqrt{\frac{\sum_{i=1}^{n} \frac{V_i}{p_{[i,n[}} + C + \left(\frac{1}{p_{[1,n[}} - 1\right) R}{\lambda \boldsymbol{\alpha}^T M \boldsymbol{\alpha}}}. \tag{1.4}$$

In that case, the overhead is given by

$$H(W^*) = 2\sqrt{\lambda \boldsymbol{\alpha}^T M \boldsymbol{\alpha} \left(\sum_{i=1}^{n} \frac{V_i}{p_{[i,n[}} + C + \left(\frac{1}{p_{[1,n[}} - 1\right) R \right)} + \sum_{i=1}^{n} \left(\frac{1}{p_{[i,n[}} - 1\right) \alpha_i + o(\sqrt{\lambda}). \tag{1.5}$$

Proof. From the expected execution time of a pattern given in Equation (1.1), we can derive the overhead as follows:

$$H(W) = \frac{\mathbb{E}(W)}{W} - 1 = \frac{\sum_{i=1}^{n} \frac{V_i}{p_{[i,n[}} + C + \left(\frac{1}{p_{[1,n[}} - 1\right) R}{W} + \lambda W \boldsymbol{\alpha}^T M \boldsymbol{\alpha} + \sum_{i=1}^{n} \left(\frac{1}{p_{[i,n[}} - 1\right) \alpha_i + \lambda \left(\frac{R}{p_{[1,n[}} + \boldsymbol{\alpha}^T M \mathbf{v} \right) + o(\lambda). \tag{1.6}$$

The optimal pattern length that minimizes the execution overhead can now be computed by balancing the first two terms of the above equation, which gives rise to Equation (1.4). Now, substituting W^* back into Equation (1.6), we can obtain the execution overhead shown in Equation (1.5). Note that when the platform MTBF $\mu = 1/\lambda$ is large in front of the resilience parameters, the last two terms of Equation (1.6) become negligible compared to other dominating terms given in Equation (1.5), so they are absorbed into $o(\sqrt{\lambda})$. □

1.5.2 Usefulness of imprecise detectors

We now assess the usefulness of partial detectors with imperfect precision. We show that an imprecise partial verification (i.e., with $p < 1$) is not used in the optimal pattern. The result is valid when the platform MTBF $\mu = 1/\lambda$ is large in front of the resilience parameters, and when the precision values are constants and independent of the error rate λ.

Theorem 2. *The optimal pattern contains no detector with imprecise verification.*

Proof. We show that given any pattern containing imprecise verifications, we can transform it into one that does not use any imprecise verification and that has a better execution overhead.

Consider a given pattern $\text{PATTERN}(W, n, \boldsymbol{\alpha}, \mathbf{D})$ that contains imprecise verifications. Theorem 1 gives the optimal length of the pattern as well as the execution overhead in that case. From Equation (1.5), we observe that the overhead is dominated by the term $\sum_{i=1}^{n} \left(\frac{1}{p_{[i,n[}} - 1\right) \alpha_i$, if the precisions of all detectors are constants and independent of the error rate λ. Assuming that the size of each segment in the pattern is also a constant fraction of the pattern length, we can improve the overhead by making α_i approach 0 for all segment i with $p_{[i,n[} < 1$. Suppose segment m is the first segment that satisfies $p_{[m,n[} = 1$. Then the execution overhead of the pattern becomes

$$H = 2\sqrt{\lambda \boldsymbol{\alpha}^T M \boldsymbol{\alpha} \left(\sum_{i=1}^{n} \frac{V_i}{p_{[i,n[}} + C + \left(\frac{1}{p_{[1,n[}} - 1 \right) R \right)} + o(\sqrt{\lambda}),$$

where $\boldsymbol{\alpha} = [0, \ldots, 0, \alpha_m, \ldots, \alpha_n]^T$. Now, by removing the first $m-1$ detectors while keeping the relative sizes of the remaining segments unchanged, we get a new pattern whose overhead is

$$H' = 2\sqrt{\lambda \boldsymbol{\alpha}'^T M' \boldsymbol{\alpha}' \left(\sum_{i=m}^{n} V_i + C \right)} + o(\sqrt{\lambda}),$$

where $\boldsymbol{\alpha}' = [\alpha_m, \ldots, \alpha_n]^T$ and M' is the submatrix of M by removing the first $m-1$ rows and columns. Clearly, we have $H' < H$ since $\sum_{i=m}^{n} V_i + C < \sum_{i=1}^{n} \frac{V_i}{p_{[i,n[}} + C + \left(\frac{1}{p_{[1,n[}} - 1\right) R$ and $\boldsymbol{\alpha}'^T M' \boldsymbol{\alpha}' = \boldsymbol{\alpha}^T M \boldsymbol{\alpha}$. □

Theorem 2 is valid up to first-order estimations and shows that an imprecise partial verification should not be used when the platform MTBF is large. Intuitively, this is because a low precision induces too much re-execution overhead when the error rate is small, making the verification unworthy. Again, we point out that this result holds when the precision can be considered as a constant, which is true in practice as the accuracy of a detector is independent of the error rate. In fact, many practical fault filters do have almost perfect precision under realistic settings [10, 31, 32]. Still, the result is striking, because it is the opposite of what is observed for predictors, for which recall matters more than precision [4].

In the rest of this chapter, we will focus on partial verifications with perfect precision (i.e., $p = 1$) but imperfect recall (i.e., $r < 1$).

1.5.3 Two key parameters

For a pattern $\text{PATTERN}(W, n, \boldsymbol{\alpha}, \mathbf{D})$ and assuming that all detectors have perfect precision, the expected execution time of the pattern according to Proposition 1 is given by

$$\mathbb{E}(W) = W + \sum_{i=1}^{n} V_i + C + \lambda W \left(R + W\boldsymbol{\alpha}^T M \boldsymbol{\alpha} + \boldsymbol{\alpha}^T M \mathbf{v}\right) + o(\lambda),$$

where M is an $n \times n$ matrix defined by $M_{ij} = 1$ for $i \leq j$ and $M_{ij} = g_{[i,j[}$ for $i > j$.

To characterize such as pattern, we introduce two key parameters in the following.

Definition 1. *The* fault-free overhead o_{ff} *of a pattern* $\text{PATTERN}(W, n, \boldsymbol{\alpha}, \mathbf{D})$ *is*

$$o_{ff} = \sum_{i=1}^{n} V_i + C, \tag{1.7}$$

and the fraction of re-executed work in case of faults is

$$f_{re} = \boldsymbol{\alpha}^T M \boldsymbol{\alpha}. \tag{1.8}$$

According to Theorem 1, we can get the optimal pattern length and execution overhead as

$$W^* = \sqrt{\frac{o_{\text{ff}}}{\lambda f_{\text{re}}}},$$
$$H(W^*) = 2\sqrt{\lambda o_{\text{ff}} f_{\text{re}}} + o(\sqrt{\lambda}).$$

The equation above shows that when the platform MTBF $\mu = 1/\lambda$ is large in front of the resilience parameters, the expected execution overhead of the optimal pattern is dominated by $2\sqrt{\lambda o_{\text{ff}} f_{\text{re}}}$. The problem is then reduced to the minimization of the product $o_{\text{ff}} f_{\text{re}}$. Intuitively, this calls for a tradeoff between fault-free overhead and fault-induced re-execution, as a smaller fault-free overhead o_{ff} tends to induce a larger re-execution fraction f_{re}, and vice versa.

1.5.4 Optimal positions of verifications

To fully characterize an optimal pattern, we have to determine its number of segments, as well as the type and position of each partial verification. In this section, we consider a pattern whose number of segments is given together with the types of all partial verifications, that is, the value of o_{ff} (Equation (1.7)) is given. We show how to determine the optimal length of each segment (or equivalently, the optimal position of each verification), so as to minimize the value of f_{re} (Equation (1.8)). The following theorem shows the result. It is the most technically involved contribution of this chapter.

Theorem 3. *Consider a pattern* $\text{PATTERN}(W, n, \boldsymbol{\alpha}, \mathbf{D})$ *where* W*,* n*, and* $\mathbf{D}$ *are given. The fraction of re-executed work* f_{re} *is minimized when* $\boldsymbol{\alpha} = \boldsymbol{\alpha}^*$*, where*

$$\alpha_k^* = \frac{1}{U_n} \cdot \frac{1 - g_{k-1} g_k}{(1 + g_{k-1})(1 + g_k)} \quad \textit{for } 1 \leq k \leq n, \tag{1.9}$$

with $g_0 = g_n = 0$ *and*

$$U_n = 1 + \sum_{i=1}^{n-1} \frac{1 - g_i}{1 + g_i}. \tag{1.10}$$

In that case, the value of f_{re} *is*

$$f_{re}^* = \frac{1}{2}\left(1 + \frac{1}{U_n}\right). \tag{1.11}$$

The goal is to minimize $f_{\text{re}} = \boldsymbol{\alpha}^T M \boldsymbol{\alpha}$ (Equation (1.8)) subject to the constraint $\sum_{k=1}^{n} \alpha_k = 1$, which we rewrite as $\mathbf{c}^T\boldsymbol{\alpha} = 1$ with $\mathbf{c} = [1, 1, \ldots, 1]^T$. Hence, we have a quadratic minimization problem under a linear constraint. For convenience, let us replace M by $A = \frac{M+M^T}{2}$, which gives the same value for f_{re}, and we obtain the symmetric matrix A defined as $A_{ij} = \frac{1+g_{[i,j[}}{2}$ for $i \leq j$. For instance, when $n = 4$, we have:

$$A = \frac{1}{2}\begin{bmatrix} 2 & 1+g_1 & 1+g_1g_2 & 1+g_1g_2g_3 \\ 1+g_1 & 2 & 1+g_2 & 1+g_2g_3 \\ 1+g_1g_2 & 1+g_2 & 2 & 1+g_3 \\ 1+g_1g_2g_3 & 1+g_2g_3 & 1+g_3 & 2 \end{bmatrix}.$$

When A is symmetric positive definite (SPD), which we will show later in the proof, there is a unique solution

$$f_{\text{re}}^{\text{opt}} = \frac{1}{\mathbf{c}^T A^{-1}\mathbf{c}}, \tag{1.12}$$

obtained for

$$\boldsymbol{\alpha}^{\text{opt}} = \frac{A^{-1}\mathbf{c}}{\mathbf{c}^T A^{-1}\mathbf{c}}. \tag{1.13}$$

This result is shown as follows. Let a *valid* vector $\boldsymbol{\alpha}$ be a vector such that $\mathbf{c}^T\boldsymbol{\alpha} = 1$. We have $\mathbf{c}^T\boldsymbol{\alpha}^{\text{opt}} = f_{\text{re}}^{\text{opt}}(\mathbf{c}^T A^{-1}\mathbf{c}) = 1$, hence $\boldsymbol{\alpha}^{\text{opt}}$ is indeed a valid vector. Then, because A is SPD, we have $X = (\boldsymbol{\alpha} - \boldsymbol{\alpha}^{\text{opt}})^T A(\boldsymbol{\alpha} - \boldsymbol{\alpha}^{\text{opt}}) \geq 0$ for any valid vector $\boldsymbol{\alpha}$, and $X = 0$ if and only if $\boldsymbol{\alpha} = \boldsymbol{\alpha}^{\text{opt}}$. Developing X, we get

$$X = \boldsymbol{\alpha}^T A\boldsymbol{\alpha} - 2\boldsymbol{\alpha}^T A\boldsymbol{\alpha}^{\text{opt}} + (\boldsymbol{\alpha}^{\text{opt}})^T A\boldsymbol{\alpha}^{\text{opt}}.$$

We have $\boldsymbol{\alpha}^T A\boldsymbol{\alpha}^{\text{opt}} = f_{\text{re}}^{\text{opt}}\boldsymbol{\alpha}^T\mathbf{c} = f_{\text{re}}^{\text{opt}}$ because $\mathbf{c}^T\boldsymbol{\alpha} = 1$. Similarly, we get $(\boldsymbol{\alpha}^{\text{opt}})^T A\boldsymbol{\alpha}^{\text{opt}} = f_{\text{re}}^{\text{opt}}$. Hence, we derive that $X = \boldsymbol{\alpha}^T A\boldsymbol{\alpha} - f_{\text{re}}^{\text{opt}} \geq 0$, with equality if and only if $\boldsymbol{\alpha} = \boldsymbol{\alpha}^{\text{opt}}$. Hence the optimal value of f_{re} is achieved for $\boldsymbol{\alpha}^{\text{opt}}$, and is equal to $f_{\text{re}}^{\text{opt}}$.

In the following, we prove that A is symmetric positive definite (SPD), and that $\boldsymbol{\alpha}^{\text{opt}} = \boldsymbol{\alpha}^*$ and $f_{\text{re}}^{\text{opt}} = f_{\text{re}}^*$. To avoid ambiguity, we use superscripts like $A^{(n)}$ whenever needed to identify the problem size n (the number of work segments).

From the definition of matrix A, we can rewrite $A^{(n)}$ as:

$$A^{(n)} = \frac{1}{2}\left(J^{(n)} + B^{(n)}\right),$$

where $J^{(n)}$ is the $n \times n$ matrix whose entries are all 1, and $B^{(n)}$ is the $n \times n$ matrix defined by $B^{(n)}_{ij} = g_{[i,j[}$ for $i \leq j$.

We start by proving two properties of $\boldsymbol{\alpha}^*$.

Lemma 1. *$\boldsymbol{\alpha}^*$ is a valid vector, i.e., $\sum_{k=1}^{n} \alpha_k^* = 1$.*

Proof. The proof is by induction on n. First, for $n = 1$ we do have $\sum_{k=1}^{1} \alpha_k^{*(1)} = 1$. For $n = 2$, we have $\sum_{k=1}^{2} \alpha_k^{*(2)} = \frac{1+g_1}{2}\left(\frac{1}{1+g_1} + \frac{1}{1+g_1}\right) = 1$, which is also correct. Assume that this result holds up to $n-1$. We can express $\boldsymbol{\alpha}^{*(n)}$ as:

$$\boldsymbol{\alpha}^{*(n)} = \frac{U_{n-1}}{U_n} \begin{bmatrix} \alpha_1^{*(n-1)} \\ \alpha_2^{*(n-1)} \\ \vdots \\ \alpha_{n-2}^{*(n-1)} \\ \alpha_{n-1}^{*(n-1)} \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ -g_{n-1}\alpha_n^{*(n)} \\ \alpha_n^{*(n)} \end{bmatrix}. \tag{1.14}$$

Therefore, we have:

$$\begin{aligned} \sum_{k=1}^{n} \alpha_k^{*(n)} &= \sum_{k=1}^{n-2} \alpha_k^{*(n)} + \alpha_{n-1}^{*(n)} + \alpha_n^{*(n)} \\ &= \frac{U_{n-1}}{U_n} \sum_{k=1}^{n-2} \alpha_k^{*(n-1)} + \frac{U_{n-1}}{U_n}\alpha_{n-1}^{*(n-1)} - g_{n-1}\alpha_n^{*(n)} + \alpha_n^{*(n)} \\ &= \frac{U_{n-1}}{U_n} \sum_{k=1}^{n-1} \alpha_k^{*(n-1)} + \alpha_n^{*(n)}(1 - g_{n-1}) \\ &= \frac{U_{n-1}}{U_n} \sum_{k=1}^{n-1} \alpha_k^{*(n-1)} + \frac{1}{U_n} \cdot \frac{1 - g_{n-1}}{1 + g_{n-1}}. \end{aligned}$$

Now, using the inductive hypothesis that $\sum_{k=1}^{n-1} \alpha_k^{*(n-1)} = 1$, we get:

$$\begin{aligned} \sum_{k=1}^{n} \alpha_k^{*(n)} &= \frac{1}{U_n}\left(U_{n-1} + \frac{1 - g_{n-1}}{1 + g_{n-1}}\right) \\ &= \frac{1}{U_n} \cdot U_n \\ &= 1, \end{aligned}$$

which concludes the proof. □

Lemma 2. $A\boldsymbol{\alpha}^* = f_{re}^* \mathbf{c}$.

Proof. We have $A = \frac{1}{2}(J + B)$ and from Lemma 1 $J\boldsymbol{\alpha}^* = \left(\sum_{k=1}^{n} \alpha_k^*\right)\mathbf{c} = \mathbf{c}$. The result will follow if we show

$$B\boldsymbol{\alpha}^* = \frac{\mathbf{c}}{U_n}, \tag{1.15}$$

for all $n \geq 1$. Equivalently, letting $\boldsymbol{\gamma} = U_n\boldsymbol{\alpha}^*$, we prove by induction on n that $B^{(n)}\boldsymbol{\gamma}^{(n)} = \mathbf{c}^{(n)}$. First, for $n = 1$ we have $B^{(1)}\boldsymbol{\gamma}^{(1)} = 1$, and for $n = 2$ we get:

$$B^{(2)}\boldsymbol{\gamma}^{(2)} = \begin{bmatrix} 1 & g_1 \\ g_1 & 1 \end{bmatrix}\begin{bmatrix} \frac{1}{1+g_1} \\ \frac{1}{1+g_1} \end{bmatrix} = \begin{bmatrix} \frac{1}{1+g_1} + \frac{g_1}{1+g_1} \\ \frac{g_1}{1+g_1} + \frac{1}{1+g_1} \end{bmatrix} = \mathbf{c}^{(2)}.$$

Now, suppose the result holds up to $n-1$. We can write:

$$\begin{aligned} B^{(n)}\boldsymbol{\gamma}^{(n)} &= \begin{bmatrix} B^{(n-1)} & \mathbf{x}^{(n-1)} \\ \left(\mathbf{x}^{(n-1)}\right)^T & 1 \end{bmatrix}\begin{bmatrix} \bar{\boldsymbol{\gamma}}^{(n-1)} \\ \gamma_n^{(n)} \end{bmatrix} \\ &= \begin{bmatrix} B^{(n-1)}\bar{\boldsymbol{\gamma}}^{(n-1)} + \mathbf{x}^{(n-1)}\gamma_n^{(n)} \\ \left(\mathbf{x}^{(n-1)}\right)^T\bar{\boldsymbol{\gamma}}^{(n-1)} + \gamma_n^{(n)} \end{bmatrix}, \end{aligned} \tag{1.16}$$

where $\bar{\boldsymbol{\gamma}}^{(n-1)}$ is the $(n-1)\times 1$ truncated vector containing the first $n-1$ elements of $\boldsymbol{\gamma}^{(n)}$ (for a problem of size n), and $\mathbf{x}^{(n-1)}$ is an $(n-1)\times 1$ vector defined as $\mathbf{x}^{(n-1)} = \begin{bmatrix} g_{[1,n-1[} & g_{[2,n-1[} & \cdots & g_{n-1} \end{bmatrix}^T$. For instance, for $n = 4$ we have $\mathbf{x}^{(3)} = \begin{bmatrix} g_1g_2g_3 & g_2g_3 & g_3 \end{bmatrix}^T$. Then the goal is to show $B^{(n-1)}\bar{\boldsymbol{\gamma}}^{(n-1)} + \mathbf{x}^{(n-1)}\gamma_n^{(n)} = \mathbf{c}^{(n-1)}$ and $\left(\mathbf{x}^{(n-1)}\right)^T\bar{\boldsymbol{\gamma}}^{(n-1)} + \gamma_n^{(n)} = 1$. From Equation (1.14), we can derive:

$$\begin{aligned} &B^{(n-1)}\bar{\boldsymbol{\gamma}}^{(n-1)} \\ &= B^{(n-1)}\left(\boldsymbol{\gamma}^{(n-1)} + \begin{bmatrix} \mathbf{0}^{(n-2)} \\ -g_{n-1}\gamma_n^{(n)} \end{bmatrix}\right) \\ &= B^{(n-1)}\boldsymbol{\gamma}^{(n-1)} + \begin{bmatrix} B^{(n-2)} & \mathbf{x}^{(n-2)} \\ \left(\mathbf{x}^{(n-2)}\right)^T & 1 \end{bmatrix}\begin{bmatrix} \mathbf{0}^{(n-2)} \\ -g_{n-1}\gamma_n^{(n)} \end{bmatrix} \\ &= B^{(n-1)}\boldsymbol{\gamma}^{(n-1)} + \begin{bmatrix} -\mathbf{x}^{(n-2)}g_{n-1}\gamma_n^{(n)} \\ -g_{n-1}\gamma_n^{(n)} \end{bmatrix} \\ &= \mathbf{c}^{(n-1)} - \mathbf{x}^{(n-1)}\gamma_n^{(n)}. \end{aligned}$$

The last line applies the inductive hypothesis $B^{(n-1)}\boldsymbol{\gamma}^{(n-1)} = \mathbf{c}^{(n-1)}$ as well as the property that $\begin{bmatrix} \mathbf{x}^{(n-2)} \\ 1 \end{bmatrix} g_{n-1} = \mathbf{x}^{(n-1)}$. Putting this result back into Equation (1.16), we derive that

$$\begin{aligned} &B^{(n-1)}\bar{\boldsymbol{\gamma}}^{(n-1)} + \mathbf{x}^{(n-1)}\gamma_n^{(n)} \\ &\quad = \mathbf{c}^{(n-1)} - \mathbf{x}^{(n-1)}\gamma_n^{(n)} + \mathbf{x}^{(n-1)}\gamma_n^{(n)} = \mathbf{c}^{(n-1)}. \end{aligned}$$

Using the property $\left(\mathbf{x}^{(n-1)}\right)^T = \left[\left(\mathbf{x}^{(n-2)}\right)^T \quad 1\right] g_{n-1}$, we can write:

$$\begin{aligned}
&\left(\mathbf{x}^{(n-1)}\right)^T \bar{\boldsymbol{\gamma}}^{(n-1)} + \gamma_n^{(n)} \\
&= \left[\left(\mathbf{x}^{(n-2)}\right)^T \quad 1\right] g_{n-1} \left(\boldsymbol{\gamma}^{(n-1)} + \begin{bmatrix} \mathbf{0}^{(n-2)} \\ -g_{n-1}\gamma_n^{(n)} \end{bmatrix}\right) + \gamma_n^{(n)} \\
&= \left[\left(\mathbf{x}^{(n-2)}\right)^T \quad 1\right] \boldsymbol{\gamma}^{(n-1)} g_{n-1} - g_{n-1}^2 \gamma_n^{(n)} + \gamma_n^{(n)}.
\end{aligned}$$

Notice that $\left[\left(\mathbf{x}^{(n-2)}\right)^T \quad 1\right] \boldsymbol{\gamma}^{(n-1)}$ is actually the last row of the product $B^{(n-1)}\boldsymbol{\gamma}^{(n-1)}$, which, by induction, is 1. Therefore we get:

$$\begin{aligned}
&\left(\mathbf{x}^{(n-1)}\right)^T \bar{\boldsymbol{\gamma}}^{(n-1)} + \gamma_n^{(n)} \\
&\quad = g_{n-1} + \gamma_n^{(n)}(1 - g_{n-1}^2) \\
&\quad = g_{n-1} + \frac{(1+g_{n-1})(1-g_{n-1})}{1+g_{n-1}} \\
&\quad = g_{n-1} + 1 - g_{n-1} \\
&\quad = 1.
\end{aligned}$$

This concludes the proof. □

We now prove that A is SPD. This requires several intermediate steps.

Lemma 3. *B is nonsingular and $\boldsymbol{\alpha}^* = \frac{1}{U_n} B^{-1}\mathbf{c}$.*

Proof. To prove that B is nonsingular, we prove by induction on n that $B^{(n)}\mathbf{y}^{(n)} = \mathbf{0}^{(n)}$ has only one solution $\mathbf{y}^{(n)} = \mathbf{0}^{(n)}$. First, for $n = 1$ we have $y_1^{(1)} = 0$, which is correct. Then, for $n = 2$ we have the following equation:

$$\begin{bmatrix} 1 & g_1 \\ g_1 & 1 \end{bmatrix} \begin{bmatrix} y_1^{(2)} \\ y_2^{(2)} \end{bmatrix} = \mathbf{0}^{(2)},$$

from which we derive $y_1^{(2)}(1 - g_1^2) = 0$ and $y_2^{(2)}(1 - g_1^2) = 0$, hence $\mathbf{y}^{(2)} = \mathbf{0}^{(2)}$, which is also correct. Now, assume that the result holds up to $n - 1$. We want to solve the general equation:

$$\begin{bmatrix} B^{(n-1)} & \mathbf{x}^{(n-1)} \\ \left(\mathbf{x}^{(n-1)}\right)^T & 1 \end{bmatrix} \begin{bmatrix} \bar{\mathbf{y}}^{(n-1)} \\ y_n^{(n)} \end{bmatrix} = \mathbf{0}^{(n)}, \tag{1.17}$$

which is equivalent to:

$$\begin{bmatrix} B^{(n-2)} & \mathbf{x}^{(n-2)} & \mathbf{x}^{(n-2)} g_{n-1} \\ \left(\mathbf{x}^{(n-2)}\right)^T & 1 & g_{n-1} \\ \left(\mathbf{x}^{(n-2)}\right)^T g_{n-1} & g_{n-1} & 1 \end{bmatrix} \begin{bmatrix} \bar{\mathbf{y}}^{(n-2)} \\ y_{n-1}^{(n)} \\ y_n^{(n)} \end{bmatrix} = \mathbf{0}^{(n)}, \tag{1.18}$$

where $\bar{\mathbf{y}}^{(n-1)}$ and $\bar{\mathbf{y}}^{(n-2)}$ are the truncated vectors containing respectively the first $n-1$ and $n-2$ elements of $\mathbf{y}^{(n)}$ (for a problem of size n). First, let us expand Equation (1.18) and consider only the last two equations of the system:

$$\left(\mathbf{x}^{(n-2)}\right)^T \bar{\mathbf{y}}^{(n-2)} + y_{n-1}^{(n)} + g_{n-1} y_n^{(n)} = 0$$
$$g_{n-1}\left(\mathbf{x}^{(n-2)}\right)^T \bar{\mathbf{y}}^{(n-2)} + g_{n-1} y_{n-1}^{(n)} + y_n^{(n)} = 0.$$

We can derive that $y_n^{(n)}(1 - g_{n-1}^2) = 0$, hence $y_n^{(n)} = 0$. Then, plugging $y_n^{(n)} = 0$ back into Equation (1.17), we derive that:

$$B^{(n-1)}\bar{\mathbf{y}}^{(n-1)} = \mathbf{0}^{(n-1)}.$$

Using the induction hypothesis for $B^{(n-1)}\bar{\mathbf{y}}^{(n-1)} = \mathbf{0}^{(n-1)}$, we have $\bar{\mathbf{y}}^{(n-1)} = \mathbf{0}^{(n-1)}$ and thus $\mathbf{y}^{(n)} = \mathbf{0}^{(n)}$, which implies that $B^{(n)}$ is nonsingular. Hence, from Equation (1.15), we can get:

$$\boldsymbol{\alpha}^* = \frac{1}{U_n} B^{-1}\mathbf{c},$$

which concludes the proof. □

Lemma 4. *A is nonsingular.*

Proof. To prove that A is nonsingular, we solve $A\mathbf{y} = \mathbf{0}$ and show that $\mathbf{y} = \mathbf{0}$. First, we can write:

$$J\mathbf{y} + B\mathbf{y} = 0,$$
$$B\mathbf{y} = -J\mathbf{y} = -\left(\sum_{i=1}^{n} y_i\right)\mathbf{c}.$$

From Lemma 3, we know that B is nonsingular and $B^{-1}\mathbf{c} = U_n\boldsymbol{\alpha}^*$. Therefore, we get:

$$\mathbf{y} = -U_n\left(\sum_{i=1}^{n} y_i\right)\boldsymbol{\alpha}^*. \tag{1.19}$$

Summing the components of both sides of Equation (1.19), we obtain:

$$\left(\sum_{i=1}^{n} y_i\right) = -U_n\left(\sum_{i=1}^{n} y_i\right)\left(\sum_{i=1}^{n} \alpha_i^*\right).$$

Since $\sum_{i=1}^{n} \alpha_i^* = 1$ from Lemma 1, we have:

$$\left(\sum_{i=1}^{n} y_i\right)(1 + U_n) = 0,$$
$$\sum_{i=1}^{n} y_i = 0,$$

which implies $\mathbf{y} = \mathbf{0}$ from Equation (1.19), and this concludes the proof that A is nonsingular. □

Lemma 5. *The last column of B^{-1} is given by:*

$$\mathbf{z} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ -g_{n-1}z_n \\ z_n \end{bmatrix}, \text{ with } z_n = \frac{1}{1 - {g_{n-1}}^2}.$$

Proof. Because we do not need the whole inverse of B, we solve $B\mathbf{z} = \mathbf{d}$, where $\mathbf{d} = \begin{bmatrix} 0 & 0 & \cdots & 1 \end{bmatrix}^T$, hence $\mathbf{z}$ will be the last column of B^{-1}. We can write:

$$\begin{bmatrix} B^{(n-2)} & \mathbf{x}^{(n-2)} & \mathbf{x}^{(n-2)}g_{n-1} \\ \left(\mathbf{x}^{(n-2)}\right)^T & 1 & g_{n-1} \\ \left(\mathbf{x}^{(n-2)}\right)^T g_{n-1} & g_{n-1} & 1 \end{bmatrix} \begin{bmatrix} \bar{\mathbf{z}}^{(n-2)} \\ z_{n-1}^{(n)} \\ z_n^{(n)} \end{bmatrix} = \mathbf{d}^{(n)},$$

where $\bar{\mathbf{z}}^{(n-2)}$ is the truncated vector containing the first $n-2$ elements of $\mathbf{z}^{(n)}$. Expanding the product, we get the following system of equations:

$$\begin{aligned} B^{(n-2)}\bar{\mathbf{z}}^{(n-2)} + \mathbf{x}^{(n-2)}z_{n-1}^{(n)} + \mathbf{x}^{(n-2)}g_{n-1}z_n^{(n)} &= \mathbf{0}^{(n-2)}, \\ \left(\mathbf{x}^{(n-2)}\right)^T \bar{\mathbf{z}}^{(n-2)} + z_{n-1}^{(n)} + g_{n-1}z_n^{(n)} &= 0, \\ \left(\mathbf{x}^{(n-2)}\right)^T g_{n-1}\bar{\mathbf{z}}^{(n-2)} + g_{n-1}z_{n-1}^{(n)} + z_n^{(n)} &= 1. \end{aligned}$$

Since $B^{(n)}$ is nonsingular, there is a unique solution. We can check that $\bar{\mathbf{z}}^{(n-2)} = \mathbf{0}^{(n-2)}$, $z_{n-1}^{(n)} = \frac{-g_{n-1}}{1-{g_{n-1}}^2}$ and $z_n^{(n)} = \frac{1}{1-{g_{n-1}}^2}$ is indeed a solution, which concludes the proof. □

Remark. The matrix B is an extension of the famous KMS symmetric matrix K [43], where $K_{ij} = g^{j-i}$ for $i \leq j$ (recall that $B_{ij} = g_{[i,j[}$). The inverse of B turns out to be tridiagonal, just as that of K, and we get:

$$B_{ij}^{-1} = \begin{cases} -\frac{g_j}{1-g_j^2} & \text{if } i = j+1 \\ -\frac{g_i}{1-g_i^2} & \text{if } i = j-1 \\ \frac{1-g_{i-1}^2 g_i^2}{(1-g_{i-1}^2)(1-g_i^2)} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}.$$

For instance, when $n = 4$ we have

$$B^{-1} = \begin{bmatrix} \frac{1}{1-g_1^2} & -\frac{g_1}{1-g_1^2} & 0 & 0 \\ -\frac{g_1}{1-g_1^2} & \frac{1-g_1^2 g_2^2}{(1-g_1^2)(1-g_2^2)} & -\frac{g_2}{1-g_2^2} & 0 \\ 0 & -\frac{g_2}{1-g_2^2} & \frac{1-g_2^2 g_3^2}{(1-g_2^2)(1-g_3^2)} & -\frac{g_3}{1-g_3^2} \\ 0 & 0 & -\frac{g_3}{1-g_3^2} & \frac{1}{1-g_3^2} \end{bmatrix}.$$

The proof of this result is very similar to the proof of Lemma 5.

Lemma 6. $A^{-1}_{nn} = 2\frac{U_n(1+g_{n-1})+2g_{n-1}}{(U_n+1)(1-g_{n-1})(1+g_{n-1})^2}$.

Proof. As in the proof of Lemma 5, we compute the last column of A^{-1}, which we call $\boldsymbol{\beta}$, by solving $A\boldsymbol{\beta} = \mathbf{d}$. Because we already solved $B\mathbf{z} = \mathbf{d}$, we have:

$$\begin{aligned} A\boldsymbol{\beta} &= B\mathbf{z} = \mathbf{d}, \\ \frac{1}{2}(J+B)\boldsymbol{\beta} &= B\mathbf{z}, \\ J\boldsymbol{\beta} &= B(2\mathbf{z} - \boldsymbol{\beta}). \end{aligned}$$

Remember that J is the matrix whose entries are all 1. Hence, we have $J\boldsymbol{\beta} = (\sum_{i=1}^n \beta_i)\,\mathbf{c}$. Also, from Lemma 3, we have $B\boldsymbol{\alpha}^* = \frac{\mathbf{c}}{U_n}$. Therefore, we can derive:

$$2\mathbf{z} - \boldsymbol{\beta} = \left(\sum_{i=1}^n \beta_i\right) U_n \boldsymbol{\alpha}^*. \tag{1.20}$$

Summing the components of both sides of Equation (1.20), we get $2\sum_{i=1}^n z_i - \sum_{i=1}^n \beta_i = (\sum_{i=1}^n \beta_i)\, U_n\, (\sum_{i=1}^n \alpha_i^*)$. Since $\sum_{i=1}^n \alpha_i^* = 1$ from Lemma 1, we get

$$\sum_{i=1}^n \beta_i = \frac{2}{U_n+1} \sum_{i=1}^n z_i.$$

From Lemma 5, we can easily compute $\sum_{i=1}^n z_i = -g_{n-1}z_n + z_n = \frac{1}{1+g_{n-1}}$. Hence, we have

$$\sum_{i=1}^n \beta_i = \frac{2}{(U_n+1)(1+g_{n-1})}.$$

Finally, from Equation (1.20), we derive that

$$\begin{aligned} \beta_n &= 2z_n - \left(\sum_{i=1}^n \beta_i\right) U_n \alpha_n^* \\ &= \frac{2}{1-{g_{n-1}}^2} - \frac{2}{(U_n+1)(1+g_{n-1})^2} \\ &= 2\frac{U_n(1+g_{n-1})+2g_{n-1}}{(U_n+1)(1-g_{n-1})(1+g_{n-1})^2}, \end{aligned}$$

which concludes the proof. □

Lemma 7. *A is symmetric positive definite (SPD).*

Proof. Note that by construction, A, J and B are all symmetric matrices. To show that A is positive definite, we show that all its principal minors are strictly positive. Recall that the principal minor of order k of $A^{(n)}$ is the determinant of the submatrix of size k that consists of the first k rows and columns of $A^{(n)}$. But this submatrix is exactly $A^{(k)}$, the matrix for the

problem of size k, so the result will follow if we show that $\det\left(A^{(n)}\right) > 0$ for all $n \geq 1$. We prove by induction on n that

$$\det\left(A^{(n)}\right) = \frac{U_n + 1}{2^n} \prod_{k=1}^{n-1} \left(1 - g_k^2\right) > 0. \tag{1.21}$$

For $n = 1$, Equation (1.21) gives $\det\left(A^{(1)}\right) = 1$, which is correct. Suppose the result holds up to $n - 1$. Since $A^{(n)}$ is nonsingular, using the co-factor method, we get that

$$\left(A^{(n)}\right)^{-1}_{nn} = \frac{\det\left(A^{(n-1)}\right)}{\det\left(A^{(n)}\right)}.$$

Therefore, using the definition of U_n and the induction hypothesis for $\det\left(A^{(n-1)}\right)$, we can get:

$$\begin{aligned}
\det\left(A^{(n)}\right) &= \frac{\det\left(A^{(n-1)}\right)}{\left(A^{(n)}\right)^{-1}_{nn}} \\
&= \frac{1}{\left(A^{(n)}\right)^{-1}_{nn}} \cdot \frac{U_{n-1} + 1}{2^{n-1}} \prod_{k=1}^{n-2} (1 - g_k^2) \\
&= \frac{1}{\left(A^{(n)}\right)^{-1}_{nn}} \cdot \frac{1}{2^{n-1}} \left(U_n - \frac{1 - g_{n-1}}{1 + g_{n-1}} + 1\right) \prod_{k=1}^{n-2} (1 - g_k^2) \\
&= \frac{1}{\left(A^{(n)}\right)^{-1}_{nn}} \cdot \frac{1}{2^{n-1}} \cdot \frac{U_n(1 + g_{n-1}) + 2g_{n-1}}{1 + g_{n-1}} \prod_{k=1}^{n-2} (1 - g_k^2).
\end{aligned} \tag{1.22}$$

Now, plugging $\left(A^{(n)}\right)^{-1}_{nn}$ from Lemma 6 into Equation (1.22), we get:

$$\begin{aligned}
\det\left(A^{(n)}\right) &= \frac{1}{2^n} \cdot \frac{(U_n + 1)(1 - g_{n-1})(1 + g_{n-1})^2}{U_n(1 + g_{n-1}) + 2g_{n-1}} \cdot \frac{U_n(1 + g_{n-1}) + 2g_{n-1}}{1 + g_{n-1}} \prod_{k=1}^{n-2} (1 - g_k^2) \\
&= \frac{U_n + 1}{2^n} (1 - g_{n-1}^2) \prod_{k=1}^{n-2} (1 - g_k^2) \\
&= \frac{U_n + 1}{2^n} \prod_{k=1}^{n-1} (1 - g_k^2),
\end{aligned}$$

which shows that Equation (1.21) holds for $\det\left(A^{(n)}\right)$ and completes the proof that $A^{(n)}$ is SPD. □

We are almost done! There remains to show that $\boldsymbol{\alpha}^{\text{opt}} = \boldsymbol{\alpha}^*$ and $f_{\text{re}}^{\text{opt}} = f_{\text{re}}^*$. But Lemma 2 shows that $A\boldsymbol{\alpha}^* = f_{\text{re}}^* \mathbf{c}$, hence $\boldsymbol{\alpha}^* = f_{\text{re}}^* A^{-1}\mathbf{c}$ and $1 = \mathbf{c}^T\boldsymbol{\alpha}^* = f_{\text{re}}^*(\mathbf{c}^T A^{-1}\mathbf{c})$, which leads to $f_{\text{re}}^{\text{opt}} = f_{\text{re}}^*$, and finally $\boldsymbol{\alpha}^{\text{opt}} = \boldsymbol{\alpha}^*$. This concludes the proof of Theorem 3.

Note that when all the partial verifications in the pattern have the same type, i.e., $g_k = g$ for all $1 \leq k < n$, obtain $f^*_{\text{re}} = \frac{1}{2}\left(1 + \frac{1+g}{n(1-g)+2g}\right)$ with

$$\alpha^*_k = \begin{cases} \frac{1}{n(1-g)+2g} & \text{for } k = 1, n \\ \frac{1-g}{n(1-g)+2g} & \text{for } k = 2, \ldots, n-1 \end{cases}.$$

The result shows that when there is only one partial verification in the pattern, i.e., $n = 2$, the two resulting segments share the same length, i.e., $\alpha_1 = \alpha_2 = \frac{1}{2}$. With two or more partial verifications of the same type, the left-most and the right-most segments, each being adjacent to a guaranteed verification, are longer than all the intermediate segments, which have the same length.

Theorem 3 also shows that, for a given set of partial verifications in a pattern, the minimum value of f_{re} does not depend upon their ordering within the pattern.

Corollary 1. *For a given set of partial verifications within a pattern, the minimum fraction of re-executed work f^*_{re} is independent of their ordering.*

1.6 Complexity

This section builds upon the previous results to provide a comprehensive complexity analysis. We introduce the *accuracy-to-cost ratio* of a detector and show that it is the key parameter to compute the optimal rational solution (Section 1.6.1). Then we establish the NP-completeness to determine the optimal integer solution (Section 1.6.2). On the positive side, we design a simple greedy algorithm whose performance is guaranteed, and sketch the construction of an FPTAS for the problem (Section 1.6.3).

1.6.1 Accuracy-to-cost ratio and rational solution

Consider a pattern $\textsc{Pattern}(W, n, \boldsymbol{\alpha}, \mathbf{D})$. Let m_j denote the number of partial verifications using detector type $D^{(j)}$ in the pattern (the number of indices $i < n$ such that D_i is of type $D^{(j)}$), and define $\mathbf{m} = [m_1, m_2, \ldots, m_k]$. Section 1.5.1 shows that minimizing the execution overhead of the pattern is equivalent to minimizing the product $o_{\text{ff}} f_{\text{re}}$. From Equations (1.7) and (1.11), we have $o_{\text{ff}} f_{\text{re}} = \frac{V^*+C}{2} f(\mathbf{m})$, where

$$f(\mathbf{m}) = \left(1 + \frac{1}{1 + \sum_{j=1}^{k} m_j a^{(j)}}\right)\left(1 + \sum_{j=1}^{k} m_j b^{(j)}\right). \tag{1.23}$$

In Equation (1.23), we define $a^{(j)} = \frac{1-g^{(j)}}{1+g^{(j)}}$ to be the *accuracy* of detector $D^{(j)}$ and define $b^{(j)} = \frac{V^{(j)}}{V^*+C}$ to be the *relative cost* of $D^{(j)}$. Furthermore, we define $\phi^{(j)} = \frac{a^{(j)}}{b^{(j)}}$ to be the *accuracy-to-cost ratio* of $D^{(j)}$. We will show that this ratio plays a key role in selecting the best detector(s).

Altogether, minimizing the pattern overhead amounts to finding the solution $\mathbf{m} = [m_1, m_2, \ldots, m_k]$ that minimizes $f(\mathbf{m})$, with $m_j \in \mathbb{N}_0$ for all $1 \leq j \leq k$. Indeed, once $\mathbf{m}$ is given, Proposition 1

and Theorem 3 completely characterize the optimal pattern, giving its length W, the number of segments $n = \sum_{j=1}^{k} m_j + 1$, and the locations $\boldsymbol{\alpha}$ of all partial detectors (whose ordering does not matter).

We first derive the optimal solution if we relax the integer constraint on $\mathbf{m}$. A rational solution in this case is denoted by $\bar{\mathbf{m}} = [\bar{m}_1, \bar{m}_2, \ldots, \bar{m}_k]$ with $\bar{m}_j \geq 0$ for all $1 \leq j \leq k$. The optimal value of $f(\bar{\mathbf{m}})$ is a lower bound on the optimal integer solution.

Lemma 8. *Suppose there are k types of detectors sorted in non-increasing order of accuracy-to-cost ratio, i.e., $\phi^{(1)} \geq \phi^{(2)} \geq \cdots \geq \phi^{(k)}$. Then,*

$$f^*(\bar{\mathbf{m}}) = \begin{cases} \left(\sqrt{\frac{1}{\phi^{(1)}}} + \sqrt{1 - \frac{1}{\phi^{(1)}}}\right)^2 & \text{if } \phi^{(1)} > 2 \\ 2 & \text{otherwise} \end{cases}.$$

Proof. First, we prove that the optimal rational solution is achieved when only the detector with the largest accuracy-to-cost ratio $\phi^{(1)}$ is used. Specifically, given any rational solution $\bar{\mathbf{m}} = [\bar{m}_1, \bar{m}_2, \ldots, \bar{m}_k]$, we show that there exists a solution $\bar{\mathbf{m}}' = [\bar{m}'_1, 0, \ldots, 0]$, which satisfies $f(\bar{\mathbf{m}}') \leq f(\bar{\mathbf{m}})$. We have

$$\begin{aligned} f(\bar{\mathbf{m}}) &= \left(1 + \frac{1}{1 + \sum_{j=1}^{k} \bar{m}_j a^{(j)}}\right)\left(1 + \sum_{j=1}^{k} \bar{m}_j b^{(j)}\right) \\ &= \left(1 + \frac{1}{1 + a^{(1)} \sum_{j=1}^{k} \frac{\bar{m}_j a^{(j)}}{a^{(1)}}}\right)\left(1 + b^{(1)} \sum_{j=1}^{k} \frac{\bar{m}_j b^{(j)}}{b^{(1)}}\right). \end{aligned} \tag{1.24}$$

Let $\bar{m}'_1 = \sum_{j=1}^{k} \frac{\bar{m}_j a^{(j)}}{a^{(1)}}$ and $\bar{n}'_1 = \sum_{j=1}^{k} \frac{\bar{m}_j b^{(j)}}{b^{(1)}}$. Since $\frac{b^{(j)}}{b^{(1)}} \geq \frac{a^{(j)}}{a^{(1)}}$ for all $1 \leq j \leq k$, we get $\bar{n}'_1 = \sum_{j=1}^{k} \frac{\bar{m}_j b^{(j)}}{b^{(1)}} \geq \sum_{j=1}^{k} \frac{\bar{m}_j a^{(j)}}{a^{(1)}} = \bar{m}'_1$. Hence, Equation (1.24) can be written as

$$\begin{aligned} f(\bar{\mathbf{m}}) &= \left(1 + \frac{1}{1 + a^{(1)} \bar{m}'_1}\right)\left(1 + b^{(1)} \bar{n}'_1\right) \\ &= \left(1 + \frac{1}{1 + a^{(1)} \bar{m}'_1}\right)\left(1 + \frac{b^{(1)} \bar{n}'_1}{\bar{m}'_1} \cdot \bar{m}'_1\right) \\ &\geq \left(1 + \frac{1}{1 + a^{(1)} \bar{m}'_1}\right)\left(1 + b^{(1)} \bar{m}'_1\right) \\ &= f(\bar{\mathbf{m}}'). \end{aligned}$$

Now, define $f(\bar{m}) = \left(1 + \frac{1}{1 + a^{(1)} \bar{m}}\right)\left(1 + b^{(1)} \bar{m}\right)$. The following derives the minimum value of $f(\bar{m})$. Differentiating $f(\bar{m})$ with respect to $\bar{m}$ and solving $\frac{\partial f(\bar{m})}{\partial \bar{m}} = 0$, we get

$$\bar{m}^* = -\frac{1}{a^{(1)}} + \sqrt{\frac{1}{a^{(1)}}\left(\frac{1}{b^{(1)}} - \frac{1}{a^{(1)}}\right)}, \tag{1.25}$$

which is positive (hence a potential solution) if $\phi^{(1)} = \frac{a^{(1)}}{b^{(1)}} > 2$. Taking the second-order derivative of $f(\bar{m})$, we get

$$\frac{\partial^2 f(\bar{m})}{\partial \bar{m}^2} = \frac{2a^{(1)}(a^{(1)} - b^{(1)})}{(a^{(1)}\bar{m} + 1)^3},$$

which is positive (hence ensures that the solution is the unique minimum) for all $\bar{m} \in [0, \infty)$ if $\phi^{(1)} = \frac{a^{(1)}}{b^{(1)}} > 1$.

Thus, when $\phi^{(1)} > 2$, the optimal solution is obtained by substituting $\bar{m}^*$ into $f(\bar{m})$, and we get

$$\begin{aligned} f(\bar{m}^*) &= \left(1 + \frac{1}{1 + a^{(1)}\bar{m}^*}\right)\left(1 + b^{(1)}\bar{m}^*\right) \\ &= \left(1 + \frac{1}{\sqrt{\phi^{(1)} - 1}}\right)\left(1 - \frac{1}{\phi^{(1)}} + \sqrt{\frac{1}{\phi^{(1)}}\left(1 - \frac{1}{\phi^{(1)}}\right)}\right) \\ &= \frac{\phi^{(1)} - 1}{\phi^{(1)}} + 2\frac{\sqrt{\phi^{(1)} - 1}}{\phi^{(1)}} + \frac{1}{\phi^{(1)}} \\ &= \left(\sqrt{\frac{1}{\phi^{(1)}}} + \sqrt{1 - \frac{1}{\phi^{(1)}}}\right)^2. \end{aligned}$$

When $\phi^{(1)} \leq 2$, the minimum value of $f(\bar{m})$ is achieved at $\bar{m} = 0$, which gives $f(0) = 2$. □

Lemma 8 shows that the optimal rational solution is achieved with only one detector, namely, the one with the highest accuracy-to-cost ratio. The optimal integer solution, however, may use more than one detector. The following shows that finding the optimal integer solution is NP-complete.

1.6.2 NP-completeness

We show that finding the optimal integer solution $\mathbf{m}$ is NP-complete, even when all detectors share the same accuracy-to-cost ratio. In particular, we consider the following decision problem.

Definition 2 (Multiple Partial Verifications (MPV))**.** *Given k detectors with the same accuracy-to-cost ratio ϕ, i.e., $\frac{a^{(j)}}{b^{(j)}} = \phi$ for all $1 \leq j \leq k$, and a bound K, is there a solution $\mathbf{m}$ that satisfies*

$$\left(1 + \frac{1}{1 + \sum_{j=1}^{k} m_j a^{(j)}}\right)\left(1 + \sum_{j=1}^{k} m_j b^{(j)}\right) \leq K? \tag{1.26}$$

Theorem 4. *The MPV problem is NP-complete.*

Proof. The MPV problem is obviously in NP. We prove the completeness by a reduction from the *Unbounded Subset Sum (USS)* problem, which is known to be NP-complete [56]. Given a multiset $S = \{s_1, s_2, \ldots, s_k\}$ of k positive integers and a positive integer I, the USS problem

asks if there exists a subset $S' \subseteq S$ whose sum is exactly I, i.e., $\sum_{j=1}^{k} m_j s_j = I$, where $m_j \in \mathbb{N}_0$. We can further assume that I/s_j is not an integer for $1 \leq j \leq k$, since otherwise we would have a trivial solution.

Given an instance of the USS problem, we construct an instance of the MPV problem with k detectors. First, choose any $\phi \in \left(2, (I/s_{\max}+1)^2+1\right)$, where $s_{\max} = \max_{j=1..k} s_j$. Then, let $\frac{a}{b} = \phi$ and $-\frac{1}{a} + \sqrt{\frac{1}{a}\left(\frac{1}{b}-\frac{1}{a}\right)} = I$, so we can get $a = \frac{\sqrt{\phi-1}-1}{I}$ and $b = \frac{\sqrt{\phi-1}-1}{\phi I}$. For each $1 \leq j \leq k$, define $a^{(j)} = s_j a$ and $b^{(j)} = s_j b$. According to the range of ϕ, we have $a^{(j)} < 1$ and $b^{(j)} < 1$ for all $1 \leq j \leq k$. Finally, let $K = \left(\sqrt{\frac{1}{\phi}} + \sqrt{1-\frac{1}{\phi}}\right)^2$.

If we use only one detector, say $D^{(j)}$, then Lemma 8 shows that Equation (1.26) is satisfied with the following unique solution:

$$\begin{aligned} m_j^* &= -\frac{1}{a^{(j)}} + \sqrt{\frac{1}{a^{(j)}}\left(\frac{1}{b^{(j)}} - \frac{1}{a^{(j)}}\right)} \\ &= \frac{1}{s_j}\left(-\frac{1}{a} + \sqrt{\frac{1}{a}\left(\frac{1}{b}-\frac{1}{a}\right)}\right) = \frac{I}{s_j}, \end{aligned}$$

which is not an integer by hypothesis, but achieves the lower bound $\left(\sqrt{\frac{1}{\phi}} + \sqrt{1-\frac{1}{\phi}}\right)^2 = K$. Now, we show that, by using multiple detectors, an integer solution to the MPV instance exists if and only if there is an integer solution to the USS instance.

($\Rightarrow$) Suppose there is an integer solution $\mathbf{m} = [m_1, m_2, \ldots, m_k]$ such that $\sum_{j=1}^{k} m_j s_j = I$. Then, by employing m_j partial verifications of detector type $D^{(j)}$ for $1 \leq j \leq k$, we get

$$\begin{aligned} &\left(1 + \frac{1}{1+\sum_{j=1}^{k} m_j a^{(j)}}\right)\left(1 + \sum_{j=1}^{k} m_j b^{(j)}\right) \\ &= \left(1 + \frac{1}{1+a\sum_{j=1}^{k} m_j s_j}\right)\left(1 + b\sum_{j=1}^{k} m_j s_j\right) \\ &= \left(1 + \frac{1}{1+aI}\right)(1+bI) \\ &= \left(\sqrt{\frac{1}{\phi}} + \sqrt{1-\frac{1}{\phi}}\right)^2 = K. \end{aligned}$$

($\Leftarrow$) Suppose there is an integer solution $\mathbf{m} = [m_1, m_2, \ldots, m_k]$ to the MPV instance such that

$$\left(1 + \frac{1}{1+\sum_{j=1}^{k} m_j a^{(j)}}\right)\left(1 + \sum_{j=1}^{k} m_j b^{(j)}\right) = K.$$

This implies

$$\left(1+\frac{1}{1+a\sum_{j=1}^{k} m_j s_j}\right)\left(1+b\sum_{j=1}^{k} m_j s_j\right) = 1+2\sqrt{\frac{1}{\phi}\left(1-\frac{1}{\phi}\right)}.$$

Let $T = \sum_{j=1}^{k} m_j s_j$. Solving T from the equation above, we get the following unique solution:

$$T = -\frac{1}{a}+\sqrt{\frac{1}{a}\left(\frac{1}{b}-\frac{1}{a}\right)} = I.$$

This completes the proof of the theorem. □

1.6.3 Greedy algorithm and FPTAS

To cope with the NP-completeness of minimizing $o_{\text{ff}} f_{\text{re}}$, there is a simple and intuitive greedy algoripartials:thm: This greedy algorithm uses only the detector with the highest accuracy-to-cost ratio $\phi^{(1)}$. We compute the optimal rational number of partial verifications $\bar{m}^*$ (from Equation (1.25)) and then round it up if it is not an integer. In Section 1.7, we show that this algorithm performs quite well in practice.

Interestingly, we can guarantee the performance of this simple algoripartials:thm: From Lemma 8, we can assume $\phi^{(1)} = \frac{a^{(1)}}{b^{(1)}} > 2$. Since $a^{(1)} < 1$, we can get $b^{(1)} < 1/2$. If the optimal fractional solution $\bar{m}^*$ given in Equation (1.25) happens to be an integer, then we get the optimal solution. Otherwise, rounding it to $\lceil \bar{m}^* \rceil$ increases the objective function $f(\mathbf{m})$ shown in Equation (1.23) by at most a factor of $\delta = 1+b^{(1)} < 3/2$. According to Equation (1.5), this gives a $\sqrt{3/2}$-approximation algorithm for minimizing the expected execution overhead (and hence the makespan).

In the following, we show that it is possible to have a fully polynomial-time approximation scheme (FPTAS), which ensures, for any $\epsilon > 0$, that the solution is within $1+\epsilon$ times the optimal, and that the running time of the algorithm is polynomial in the input size and $1/\epsilon$. To develop the FPTAS, we perform the following transformations to the problem.

First, we convert all parameters in Equation (1.23) to integers. Since $a^{(j)} = \frac{1-g^{(j)}}{1+g^{(j)}} = \frac{r^{(j)}}{2-r^{(j)}} \leq 1$ and $r^{(j)}$ is rational, we can write $a^{(j)} = \frac{p_j}{q_j}$, where p_j and q_j are positive integers with $p_j \leq q_j$. We assume that C, V^* and all the $V^{(j)}$'s are also integers. Thus, minimizing $f(\mathbf{m})$ is equivalent to minimizing the following function:

$$F(\mathbf{m}) = \left(1+\frac{L}{L+\sum_{j=1}^{k} m_j L^{(j)}}\right)\left(C+V^*+\sum_{j=1}^{k} m_j V^{(j)}\right),$$

where L denotes the least common multiple (LCM) of $q_1, q_2, \ldots, q_k$, and $L^{(j)} = \frac{p_j}{q_j} L \leq L$. Clearly, L and all the $L^{(j)}$'s can be represented by a polynomial function of the original input size.

Next, we compute an upper bound on the number of partial verifications. Observe that $F(\mathbf{0}) = 2(C + V^*)$ and $F(\mathbf{m}) \geq C + V^* + \sum_{j=1}^{k} m_j V^{(j)}$. This implies that the optimal solution must satisfy $m_j \leq \frac{C+V^*}{V^{(j)}}$ for all $1 \leq j \leq k$. Therefore, it follows that $\sum_{j=1}^{k} m_j V^{(j)} \leq k(C + V^*)$. The bound on m_j allows us to transform the unbounded problem to the 0-1 problem by providing $\lfloor \log m_j \rfloor$ additional copies of each item type j with doubling $V^{(j)}$ and $L^{(j)}$ values. This is a standard technique also used in transforming the bounded and unbounded knapsack problems to the 0-1 knapsack problem [67]. The total number of items becomes $K = \sum_{j=1}^{k} (1 + \lfloor \log m_j \rfloor) = O(k \log(C + V^*))$, which stays polynomial in the input size.

Define $\mathbf{x} = [x_1, x_2, \ldots, x_K]$, and let L_j and V_j be the value and cost of item j, respectively. We can now formulate the optimization problem as follows:

$$\begin{aligned} \text{minimize } F(\mathbf{x}) &= \left(1 + \frac{L}{L + \sum_{j=1}^{K} x_j L_j}\right)\left(C + V^* + \sum_{j=1}^{K} x_j V_j\right) \\ \text{subject to } & \sum_{j=1}^{K} x_j V_j \leq k(C + V^*) \\ & x_j \in \{0, 1\} \quad \forall j = 1, 2, \ldots, K \end{aligned}$$

and the size of all parameters is a polynomial function of the input size of the original problem. To find an FPTAS for the problem above, we adopt the technique used in [33] for designing an FPTAS for the *Maximum Density Knapsack (MDK)* problem described below.

Maximum Density Knapsack (MDK): Given a set $S = \{s_1, s_2, \ldots, s_K\}$ of K items, where each item $s_j \in S$ has a positive integer profit p_j and a positive integer weight w_j, a total capacity W, and an initial weight w_0, the MDK problem is formulated as:

$$\begin{aligned} \text{maximize } & \frac{\sum_{j=1}^{K} x_j p_j}{w_0 + \sum_{j=1}^{K} x_j w_j} \\ \text{subject to } & \sum_{j=1}^{K} x_j w_j \leq W \\ & x_j \in \{0, 1\} \quad \forall j = 1, 2, \ldots, K \end{aligned}$$

Cohen and Katzir [33] give an FPTAS for the MDK problem by using the existing FPTAS for the knapsack problem [67]. In particular, their algorithm relies on the property that, for every profit P, a minimum weight solution $\mathbf{x}$ is found such that $P(\mathbf{x}) = \sum_{j=1}^{K} x_j p_j \geq \lfloor \frac{P}{1+\epsilon'} \rfloor$, for any $\epsilon' > 0$. This immediately gives rise to an FPTAS for MDK.

We can apply the same technique to construct an FPTAS for minimizing $F(\mathbf{x})$. Let $\mathbf{x}_{opt}$ denote the optimal solution. By considering V_j as weight and L_j as profit, we can run the FPTAS for knapsack and return in polynomial time a solution $\mathbf{x}$ that satisfies $P(\mathbf{x}) \geq \lfloor \frac{P(\mathbf{x}_{opt})}{1+\epsilon'} \rfloor$ and $W(\mathbf{x}) \leq W(\mathbf{x}_{opt})$. By setting carefully the value of ϵ' as a function of ϵ, the solution yields $F(\mathbf{x}) \leq (1 + \epsilon) F(\mathbf{x}_{opt})$. The detail is similar to the one presented in [33] and is omitted here.

1.7 Performance evaluation

In this section, we assess the benefits of partial detectors and evaluate the performance improvement they can provide. Both Maple-based evaluations using the performance model and realistic simulations using fault-injection are conducted. We consider four scenarios. In the first scenario, we study the optimal algorithm using only a single detector type. In the second scenario, we study the impact of the number of partial verifications on the overhead and the optimal pattern length. The third scenario tackles applications with various datasets that expose a range of recall values for each detector rather than a single value. Finally, in the fourth scenario, we focus on the greedy algorithm and compare its performance with the optimal solution that uses more than one type of partial detectors. The simulator code is available for download at `http://graal.ens-lyon.fr/~yrobert/two-level.zip`, so that interested readers can experiment with it and build relevant scenarios of their choice.

1.7.1 Simulation setup

We have chosen realistic parameters that depict a typical future exascale platform. The target platform consists of 10^5 nodes whose individual MTBF is 100 years, which amounts to a platform MTBF of $\mu = 31536$ seconds (i.e., about 8.7 hours). The global size of the memory for an exascale machine is expected to be between 32 PB and 64 PB; divided by the number of nodes (10^5), the memory size per node goes from 320 GB to 640 GB. Most HPC applications try to populate 90% of the node memory but only $10\% - 50\%$ of the memory is checkpointed. That makes the checkpoint size between 30 GB and 300 GB. At exascale, most checkpoints will be done in local non-volatile memory (NVM), which is known to be slower than DRAM. We assume checkpoint throughput between 0.5 GB/s and 1 GB/s. While the results presented in this chapter are based on these given parameters, we encourage the readers to validate the model with different architecture characteristics.

Concerning the detectors, we assume that they have an almost perfect precision, otherwise we would not use them, as shown in Section 1.5.2. Thus, the detectors will be configured to adapt their prediction error in order to minimize the number of false positives (i.e., maximize precision) at the cost of some recall. Previous studies [38] have shown that such configuration can lead to different levels of recall depending on the prediction method and the dataset behavior. Nevertheless, this large study with over 20 different types of simulations showed some trends in performance and efficacy, and we base our simulation parameters in those results. The first detector $D^{(1)}$ has a throughput of about 200 MB/s/process and a recall of 0.5 [9, 11]. The second one $D^{(2)}$ has a throughput of about 20 MB/s/process and a recall of 0.95 [15]. If we assume 512 processes per node at exascale, then the node throughput of the detectors becomes 100 GB/s for $D^{(1)}$ and 10 GB/s for $D^{(2)}$. Finally, we assume a third detector $D^{(3)}$, which is an optimized version that combines the features of the first two detectors, achieving a recall of 0.8 and a throughput of 50 GB/s. Concerning the perfect detector D^*, we assume a throughput of 0.5 GB/s based on the fact that application-specific detectors are usually based on physical properties such as mass or energy conservation, which requires global communications and is therefore more expensive than purely local checks.

The simulator generates errors following an exponential distribution of parameter λ. An

Table I
CHARACTERISTICS OF ALL DETECTOR TYPES AND THE PERFORMANCE OF THE OPTIMAL PATTERN USING EACH DETECTOR TYPE ALONE.

	$D^{(1)}$	$D^{(2)}$	$D^{(3)}$	D^*
Cost V (seconds)	3	30	6	600
Recall r	0.5	0.95	0.8	1
Accuracy-to-cost ratio ϕ	133	36	133	2
Predicted overhead H^*	29.872%	31.798%	29.872%	39.014%
Optimal W^* (hours)	2.41	2.38	2.41	1.71
Optimal m^*	33	6	17	0
Simulated overhead	30.313%	32.537%	30.743%	40.414%
Ave. # checkpoints (per day)	7.28	7.23	7.25	9.50
Ave. # recoveries (per day)	2.26	2.25	2.33	1.94

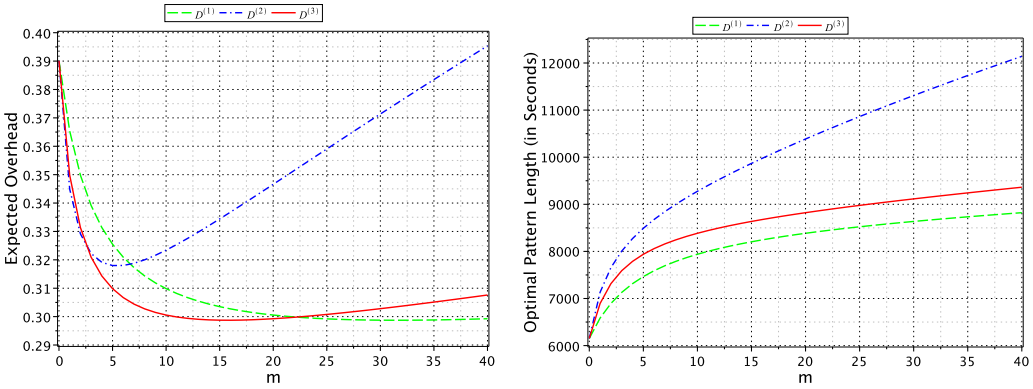
experiment goes as follows. We feed the simulator with the description of the platform consisting of the parameters described above. For each pattern, we derive the (near) optimal pattern by computing the pattern length W^*, and the optimal number m^* and positions $\boldsymbol{\alpha}^*$ of verifications, using the formulas from our model. The total amount of work for the application is then set to that of 1000 optimal patterns, i.e., $W_{\text{base}} = 1000W^*$. The simulator runs each experiment 1000 times, and the simulated overhead is obtained by averaging the results from the 1000 runs.

1.7.2 Scenario 1: Performance of different detectors

In the first scenario, we study the optimal algorithm when using a single detector type. Three detectors $D^{(1)}$, $D^{(2)}$ and $D^{(3)}$ are used separately, with respective costs and recall values $V^{(1)} = 3$ seconds, $V^{(2)} = 30$ seconds, $V^{(3)} = 6$ seconds and $r^{(1)} = 0.5$, $r^{(2)} = 0.95$, $r^{(3)} = 0.8$. The checkpointing cost and the perfect detector cost with recall $r^* = 1$ are fixed at $C = V^* = 600$ seconds.

Table I summarizes the characteristics of all detector types including the perfect detector, and presents the predicted performance of the optimal pattern using each detector alone. Recall that the accuracy-to-cost ratio is defined as $\phi^{(j)} = \frac{a^{(j)}}{b^{(j)}}$, where $a^{(j)} = \frac{r^{(j)}}{2-r^{(j)}}$ denotes the accuracy of the detector and $b^{(j)} = \frac{V^{(j)}}{V^*+C}$ the relative cost. Thanks to the higher accuracy-to-cost ratios, the use of partial verifications yields much better performance compared to the baseline algorithm that uses only guaranteed verification. In particular, $D^{(1)}$ and $D^{(3)}$, which have the highest accuracy-to-cost ratio, offer about 10% improvement in the execution overhead. This translates to about 1 hour of saving for every 10 hours of execution, which is significant in terms of cost and resource usage. The optimal pattern also employs a larger number m^* of partial verifications, due to their lower costs, so that checkpoints can be taken less frequently (i.e., with a larger period W^*).

It is interesting to observe, for $D^{(1)}$ and $D^{(3)}$, that the product of cost and frequency (number of verifications to use in a pattern) is roughly equal. Indeed, for detectors with the same accuracy-to-cost ratio ϕ, our analysis shows that the cost-frequency product is in fact a constant,



(a) (b)

Figure 1.2: Expected overhead (a) and optimal length (b) of a pattern against the number of partial verifications when a single type of detector is used for three different detectors. The costs and recalls of the three detectors are $V^{(1)} = 3$ seconds, $V^{(2)} = 30$ seconds, $V^{(3)} = 6$ seconds, and $r^{(1)} = 0.5$, $r^{(2)} = 0.95$, $r^{(3)} = 0.8$. The costs of checkpointing and guaranteed verification are $C = V^* = 600$ seconds. The platform MTBF is $\mu = 31536$ seconds.

and it is given by $V\bar{m}^* = V\left(-\frac{1}{a} + \sqrt{\frac{1}{a}\left(\frac{1}{b} - \frac{1}{a}\right)}\right) = (V^* + C)\left(-\frac{1}{\phi} + \sqrt{\frac{1}{\phi}\left(1 - \frac{1}{\phi}\right)}\right)$.

To validate the predicted performance, we have simulated the execution of the optimal patterns by injecting faults with the specified error rate. The last part of Table I shows the simulation results, obtained by averaging the values over 1000 runs for the respective patterns. We can see that the simulated overheads are within 1% of the predicted values for all patterns, which demonstrates the high accuracy of first-order approximation to the performance model. The results also confirm the low checkpointing frequency and high recovery rate of computing patterns that employ partial verifications. Intuitively, a higher recovery rate means that more errors are detected earlier in the execution. The results nicely corroborate the theoretical analysis, and demonstrate the benefit of using low-cost partial verifications for dealing with silent errors.

Since the results for realistic simulations with fault injections are very close to the model's predictions, we will focus on studying the model in the following experiments.

1.7.3 Scenario 2: Impact of number of partial verifications

In the second scenario, we study the impact of the number of partial verifications on the execution overhead and pattern length of the optimal partial verification algoripartials:thm:

Figure 1.2 plots the overhead as well as the optimal pattern length as functions of the number of partial verifications m for each detector. The plots also show the overhead ($\approx 39\%$) and optimal pattern length (≈ 6156 seconds) of the baseline algorithm, represented by $m = 0$. We can see that the expected overhead is reduced for all three detectors by employing partial verifications. For each detector, the optimal overhead is attained for a particular value of m, corroborating the theoretical study. After this point, it starts rising again due to the fact that

forcing too many verifications will eventually cause the error-free overhead to increase. The improvement in overhead over the baseline algorithm is 9% for detectors $D^{(1)}$ and $D^{(3)}$ (optimal overhead for both is $\approx 30\%$), and 7% for detector $D^{(2)}$ (optimal overhead is $\approx 32\%$).

Also, the optimal pattern length increases as more partial verifications are employed inside the pattern. This is because the use of intermediate verifications allows silent errors to be detected earlier in the pattern and thus delays the checkpointing process. Interestingly, the optimal pattern lengths of all three detectors are around 8600 seconds when their respective optimal overheads are reached. This implies that an optimal pattern using partial verifications delays the taking of each checkpoint by ≈ 40 minutes, which corresponds to a saving of ≈ 4 checkpoints/day over the baseline algoripartials:thm: Concerning the performance of detectors, we can see that $D^{(1)}$ and $D^{(3)}$ are slightly better than $D^{(2)}$, due to their higher accuracy-to-cost ratios. However, for $m \leq 2$, $D^{(2)}$ is better due to its higher recall, while its performance degrades as more $D^{(2)}$ detectors are employed due to its high cost.

1.7.4 Scenario 3: Impact of detector recall

In the third scenario, we consider applications with various datasets that expose a change in the detection recall. Therefore, a range of recall values is possible for each detector rather than a single value.

According to [8, 11], the recall ranges of the three detectors are $r^{(1)} = [0.5, 0.9]$, $r^{(2)} = [0.75, 0.95]$, and $r^{(3)} = [0.8, 0.99]$, respectively. Given a dataset, we obtain a value of recall for each detector within the range. This is because different datasets might expose different levels of entropy and therefore the detectors might expose different prediction accuracies, hence different recalls. Note that although the recall might be different for different datasets, the work done, hence the detection cost, is the same. We rely upon four different metrics, namely, optimal overhead, optimal pattern length, optimal number of verifications, and accuracy-to-cost-ratio, to assess the impact of recall r on the optimal partial verification algoripartials:thm:

Figure 1.3 compares the performance of the three detectors through the four metrics when there is a change in the detection recall for each detector in its recall range. The plots in Figure 1.3(a) show variations in the optimal overheads with increasing recall values. As expected, the optimal overheads are reduced for all three detectors, since a higher recall value for the same cost (and same number) of verification reduces the fault-induced re-execution cost (f_{re}), while keeping the fault-free overhead (o_{ff}) constant, thus minimizes the product $o_{\text{ff}} f_{\text{re}}$ (see Section 1.5.1). This reduction in overhead can also be explained through the plots in Figure 1.3(d), which show an increase in the accuracy-to-cost ratio of each detector with higher recall values. The detectors $D^{(1)}$ and $D^{(3)}$ have the highest accuracy-to-cost ratio, and when used alone inside the pattern, produce the lowest optimal overheads for their respective recall ranges. This substantiates the theoretical analysis of Lemma 8 in Section 1.6.1. The detector $D^{(2)}$, being an expensive verification, has a much lower ratio and thus incurs a higher optimal overhead.

Figure 1.3(b) shows oscillations in the curves representing the optimal pattern length for varying recall values. This can be understood by observing the plot in Figure 1.3(c), where the optimal number of partial verifications m^* for all three detectors follows a staircase function. For example, the optimal m^* for detector $D^{(1)}$ goes from 33 to 22 as the recall value increases in the range. This is due to the fact that verifications with higher recalls (or accuracy-to-cost ra-

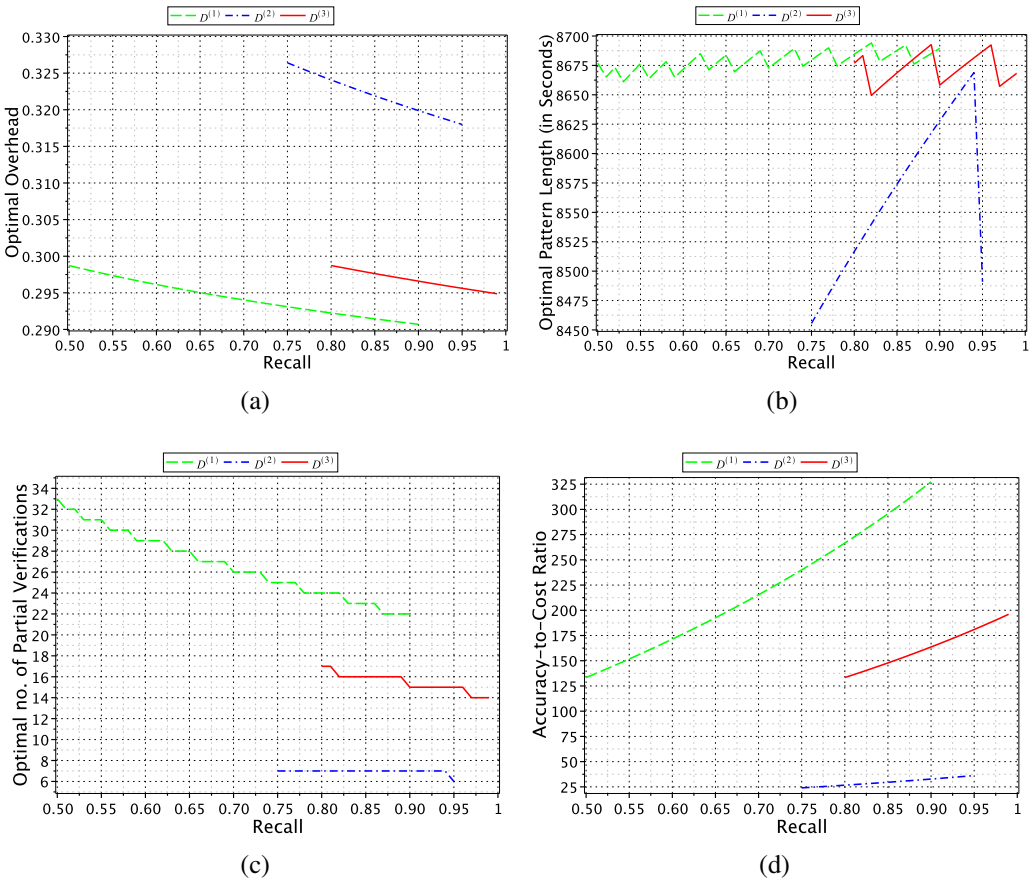


Figure 1.3: Optimal overhead (a), optimal pattern length (b), optimal number of partial verifications (c), and accuracy-to-cost ratio (d) for three different detectors as functions of recall in their respective recall ranges ($r^{(1)} = [0.5, 0.9]$, $r^{(2)} = [0.75, 0.95]$ and $r^{(3)} = [0.8, 0.99]$).

tios) allow us to achieve lower optimal overhead (as in Figure 1.3(a)) with fewer verifications. This step-wise reduction in the number of verifications leads to minor oscillations in the optimal pattern length. In particular, it is interesting to observe that in case of detector $D^{(2)}$, by fixing its recall at $r^{(2)} = 0.94$ and $r^{(2)} = 0.95$, the optimal overheads are 31.83% and 31.79% respectively, and the optimal pattern lengths are 8668 and 8490 seconds respectively. Thus, approximately 3 minutes of more execution per pattern can be done by compromising 0.04% of overhead. The reduction in the optimal pattern length for a higher recall value is due to a decrement in m^*. Note that both oscillations and staircase effects would disappear if m^* was allowed to take rational values.

1.7.5 Scenario 4: Performance of greedy algorithm

Finally, in the last scenario, we focus on the greedy algorithm presented in Section 1.6.3 and compare its performance with the optimal solution that uses more than one type of partial detector with different datasets, while keeping the same values for C, V^* and μ.

Table II
PERFORMANCE COMPARISON OF THE GREEDY ALGORITHM AND THE OPTIMAL SOLUTION. IN ALL SCENARIOS, $C = V^* = 600$ SECONDS, $V^{(1)} = 3$ SECONDS, $V^{(3)} = 6$ SECONDS.

	$\mathbf{m}$	overhead H	diff. from opt.
Scenario 1: $r^{(1)} = 0.51$, $r^{(3)} = 0.82$, $\phi^{(1)} \approx 137$, $\phi^{(3)} \approx 139$			
Optimal solution	(1, 15)	29.828%	0%
Greedy with $D^{(3)}$	(0, 16)	29.829%	0.001%
Scenario 2: $r^{(1)} = 0.58$, $r^{(3)} = 0.9$, $\phi^{(1)} \approx 163$, $\phi^{(3)} \approx 164$			
Optimal solution	(1, 14)	29.659%	0%
Greedy with $D^{(3)}$	(0, 15)	29.661%	0.002%
Scenario 3: $r^{(1)} = 0.64$, $r^{(3)} = 0.97$, $\phi^{(1)} \approx 188$, $\phi^{(3)} \approx 188$			
Optimal solution	(1, 13)	29.523%	0%
Greedy with $D^{(1)}$	(27, 0)	29.524%	0.001%
Greedy with $D^{(3)}$	(0, 14)	29.525%	0.002%

As in the previous experiment, the recall of each detector is given a range of possible values, and its actual value depends on the dataset. As shown in Figure 1.3(d), even with the recall ranges, $D^{(2)}$ always has a lower accuracy-to-cost ratio compared to $D^{(1)}$ and $D^{(3)}$, which share similar ratios. Table II presents three scenarios that we have identified, where a combination of $D^{(1)}$ and $D^{(3)}$ constitutes the optimal pattern. In all these scenarios, the greedy algorithm, which uses only the detector with the highest accuracy-to-cost ratio, performs within 0.002% of the optimal solution. The results show that the greedy algorithm performs extremely well under these practical settings, even though the optimal pattern may employ both $D^{(1)}$ and $D^{(3)}$ in the solution.

1.8 Conclusion

In this chapter, we provided a comprehensive analysis of computing patterns that employ different types of partial verifications for detecting silent errors in HPC applications. We demonstrated that detectors with imperfect precision should not be used in such computing patterns. When considering detectors with imperfect recall, we showed that the optimization problem is NP-complete in general, and we proposed both a greedy algorithm and an FPTAS for choosing the number of detectors to be used, as well as their types and locations in the pattern. Extensive simulations based on realistic detector settings showed that the greedy algorithm works well in practice, and confirmed the usefulness of partial detectors to cope with silent errors in exascale systems.

Chapter 2

Optimal Resilience Patterns with Fail-Stop and Silent Errors

Based on the results obtained in Chapter 1, as well as in a preliminary analysis [W4, J3], this chapter presents a unified framework and optimal algorithmic solutions addressing both fail-stop and silent errors. Silent errors are handled via verification mechanisms (either partially or fully accurate) and fast in-memory checkpoints. Fail-stop errors are processed via slower disk checkpoints. All verification and checkpoint types are combined into computational patterns. We provide a unified model, and a full characterization of the optimal pattern. Our results nicely extend several published solutions, and demonstrate how to make use of different techniques to solve the double threat of fail-stop and silent errors. Extensive simulations based on real data confirm the accuracy of the model, and show that patterns that combine all resilience mechanisms are required to provide acceptable overheads. The work in this chapter has been published in the proceedings of the *International Parallel & Distributed Processing Symposium* (IPDPS) [C2].

2.1 Introduction

While the detection of silent errors seriously complicates the design of resilience protocols, we have already shed some light on the use of partial verifications in Chapter 1. In particular, we have proven that the optimal pattern does not contain any imprecise detector, and that using only the detector with the highest *accuracy-to-cost-ratio* is a good approximation in practice. However, new problems arise when we consider both fail-stop and silent errors. On the one hand, silent errors naturally call for fast in-memory checkpointing, because a local copy of the data can still be used after corruption has been detected. On the other hand, fail-stop errors require to store the checkpoints on remote stable storage (disks) because the whole memory content can be lost when such a failure strikes. Granted, multi-level checkpointing protocols have been designed for several years, but we face two major difficulties when combining fail-stop and silent errors.

First, and to the best of our knowledge, the interplay of verification mechanisms with two types of checkpoints, in-memory and disk-based, has never been investigated. Second, the

inherent detection latency of silent errors renders the problem quite different from traditional multi-level checkpointing, where each failure, regardless of its level, is detected immediately upon striking. In this work, after some quite technically involved derivations, we provide the optimal solution to the problem, either with guaranteed or with partial verifications.

Our approach to solving the double problem of fail-stop and silent errors is to partition the execution of the application into *periodic patterns*, i.e., computational units that repeat over time. Each pattern ends with a guaranteed verification, an in-memory checkpoint and a disk checkpoint, so that errors do not propagate from a given pattern to the next one. Inside each pattern, there are several segments, each ending with a guaranteed verification and an in-memory checkpoint. In turn, each segment is partitioned into work chunks (possibly of different size) that are separated by partial verifications. See Figure 2.2 for an example with three segments and a total of six chunks. Several parameters should be given to fully characterize a pattern, namely the number of segments, and the number and size of each chunk inside each segment. The shape of a pattern is quite flexible, which enables us to provide the first model including two levels of checkpoints.

The main objective is to design an optimal pattern. Informally, consider a pattern P that includes W units of work (the cumulated size of all the chunks within the pattern). Without loss of generality, assume unit speed computation, so that we can speak of time or work interchangeably. In the presence of fail-stop or silent errors, the expected execution time of the pattern will be $\mathbb{E}(\mathrm{P})$: we have to take expectations, as the computation time is no longer deterministic. Note that $\mathbb{E}(\mathrm{P}) > W$ for two reasons: the time spent in checkpoints and verifications, even if there is no error, and the time lost due to recovery and re-execution after an error. An optimal pattern is defined as the one minimizing the ratio $\frac{\mathbb{E}(\mathrm{P})}{W}$, or equivalently the ratio $\frac{\mathbb{E}(\mathrm{P})-W}{W} = \frac{\mathbb{E}(\mathrm{P})}{W} - 1$. This latter ratio is the relative overhead paid for executing the pattern. The smaller this overhead, the faster the progress of the execution.

The main contributions of this work are the following:

- The design of a detailed model based upon the computational patterns described above (see Section 2.2).
- The determination of the optimal pattern, first in some particular cases (one-chunk segments, one segment with multiple chunks), and then in the general case. The comprehensive list of results summarized in Table I extends and unifies many results from the literature (see the discussion in Section 2.7).
- An extensive set of simulations that use data collected on real platforms, and extrapolate them to exascale platforms. The results confirm the accuracy of the model, as long as the MTBF is large enough in front of the resilience parameters. They also help assess the impact of each resilience mechanism, and show that patterns that combine all mechanisms (partial and guaranteed verifications and two checkpoint types) are required to provide acceptable overheads.

The rest of the chapter is organized as follows. Section 2.2 introduces the model and notation. The following sections show how to determine the optimal pattern. We start with the simplest pattern (a single one-chunk segment) in Section 2.3, extending Young and Daly's formula

to two error sources. We discuss patterns with multiple one-chunk segments in Section 2.4.1, patterns with one multiple-chunk segment in Section 2.4.2, and finally the most general pattern in Section 2.4.3. Section 2.5 deals with errors during checkpoints and recoveries. Simulation results are presented in Section 2.6. Section 2.7 surveys related work. Finally, Section 2.8 provides concluding remarks.

2.2 Model

2.2.1 Failure model

We consider a realistic scenario in large-scale systems, where hardware faults and silent data corruptions coexist. They are commonly referred to as *fail-stop* errors and *silent* errors in the literature. Since these two types of errors are caused by different sources, we assume that they are independent and that both occurrences follow a *Poisson process* with arrival rates λ_f and λ_s, respectively. Hence, the probability of having at least a fail-stop error during a computation of length w is given by $p^f = 1 - e^{-\lambda_f w}$ and the probability of having at least a silent error during the same computation is $p^s = 1 - e^{-\lambda_s w}$. We also assume that both error rates are in the same order, i.e., $\lambda_f = \Theta(\lambda)$, and $\lambda_s = \Theta(\lambda)$, where $\lambda = \lambda_f + \lambda_s = 1/\mu$ denotes the reciprocal of the platform MTBF μ while accounting for both types of failures.

2.2.2 Two-level checkpointing

To deal with both fail-stop and silent errors, resilience is provided through the use of a two-level checkpointing scheme coupled with an error detection (or verification) mechanism. The protocol is enforced by a periodic computing *pattern* as discussed in Section 2.1. When a fail-stop error strikes inside a pattern, the computation is interrupted immediately due to a hardware fault, so all the memory content is destroyed. In this case, we roll back to the beginning of the pattern and recover from the last disk checkpoint (taken at the end of the previous pattern, or the initial data for the first pattern). On the contrary, when a silent error is detected inside a pattern, either by a partial verification or by a guaranteed one, we roll back to the nearest memory checkpoint in the pattern and recover from the memory copy there, which is much cheaper than recovering from the last disk checkpoint.

We enforce the following two properties for a pattern:

- *A memory checkpoint is always taken immediately before each disk checkpoint*. Since performing an I/O operation requires first flushing the data to a memory buffer, this process incurs little extra overhead and hence has a natural justification. Indeed, such a property has been enforced in some practical multi-level checkpointing systems [12]. Similarly, when we recover from a disk checkpoint, we also restore the corresponding memory copy, which was destroyed due to the last fail-stop error.

- *A guaranteed verification is always executed immediately before each memory checkpoint*. Since storing a checkpoint can be expensive even for the memory, this property guarantees that all (memory and disk) checkpoints are valid, and hence avoids the need of

maintaining multiple checkpoints, which is known to be difficult to recover from (one has to decide which checkpoint is valid, etc.). With this property, only one memory checkpoint and one disk checkpoint need to be maintained at any time during the execution of the application.

To simplify the analysis, we assume in Sections 2.3 and 2.4 that errors only strike the computations, while verifications, memory copies, and I/O transfers are protected from failures. In Section 2.5, we show how this assumption can be relaxed in the analysis.

2.2.3 Notation

Let C_D denote the cost of disk checkpointing, C_M the cost of memory checkpointing, R_D the cost of disk recovery, and R_M the cost of memory recovery. Recall that when a disk recovery is done, we also need to restore the memory state, hence a cost $R_D + R_M$ is paid.

Also, let V^* denote the cost of guaranteed verification and V the cost of a partial verification. The partial verification is also characterized by its *recall*, which is denoted by r and represents the proportion of detected errors over all silent errors that have occurred during the execution. If multiple partial verifications are available, Chapter 1 suggests to use the one with the largest *accuracy-to-cost* ratio, which is defined as $\frac{r}{2-r}/\frac{V}{V^*+C_M}$. Note that the guaranteed verification can be considered as one with recall $r^* = 1$ and hence an accuracy-to-cost ratio $\frac{C_M}{V^*} + 1$. Since a partial verification usually incurs a much smaller cost yet has a reasonable recall, its accuracy-to-cost ratio can be orders of magnitude (e.g., 100x) better than that of the guaranteed verification [10, 15]. This characteristic makes partial verification a highly attractive technique for detecting silent errors. Hence, we make use of partial verifications between memory checkpoints in the pattern.

For clarity, we refer to the computation between any two consecutive memory checkpoints as a *segment*, and refer to the computation between two consecutive verifications as a *chunk*. Formally, a pattern $\mathrm{P}(W, n, \boldsymbol{\alpha}, \boldsymbol{m}, \langle \boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_n \rangle)$ is defined by the following parameters:

- W: total amount of computation (or work) of the pattern.
- n: number of memory checkpoints inside the pattern (also number of computational segments within the pattern).
- $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \ldots, \alpha_n]$: proportion of the segment sizes, i.e., $\alpha_i = w_i/W$, where w_i denotes the amount of work in the i-th segment of the pattern. Hence, we have $\sum_{i=1}^{n} \alpha_i = 1$.
- $\boldsymbol{m} = [m_1, m_2, \ldots, m_n]$: number of verifications inside each segment of the pattern (also number of chunks in that segment).
- $\boldsymbol{\beta}_i = [\beta_{i,1}, \beta_{i,2}, \ldots, \beta_{i,m_i}]$ $\forall i = 1, 2, \ldots, n$: proportion of the chunk sizes in the segments, i.e., $\beta_{i,j} = w_{i,j}/w_i$, where $w_{i,j}$ denotes the amount of work in the j-th chunk of the i-th segment of the pattern. Hence, we have $\sum_{j=1}^{m_i} \beta_{i,j} = 1$ for all $i = 1, 2, \ldots, n$.

The simplest pattern is illustrated in Figure 2.1, and consists of a single segment ($n = 1$, $W = w_1$) , which comprises a single chunk ($\boldsymbol{m} = [1]$). By construction, this chunk is followed by a guaranteed verification, followed immediately by a memory checkpoint and a disk checkpoint. With our notations, this pattern is denoted as $\mathrm{P}(W, 1, [1], [1], \langle [1] \rangle)$, or P_D (only disk checkpoints, which are always preceded by a guaranteed verification and a memory checkpoint).

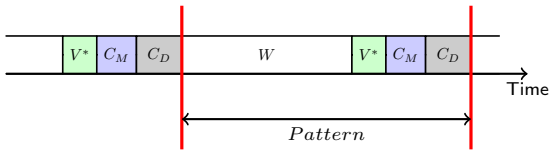


Figure 2.1: Pattern $\mathrm{P}_D = \mathrm{P}(W, 1, [1], [1], \langle [1] \rangle)$.

Figure 2.2 shows a more complicated pattern, with three segments. The first segment has three chunks, the second segment has one chunk, and the third segment has two chunks. Therefore, if a silent error is detected by the guaranteed verification at the end of the second segment, it is possible to recover from the memory checkpoint preceding it, rather than starting the whole pattern again. Additionally, silent errors may be detected earlier in the first and third segment thanks to the additional partial verifications.

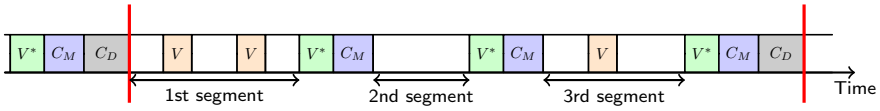


Figure 2.2: Pattern with three segments and six chunks.

2.2.4 Objective

The objective is to find a pattern that minimizes the expected execution time of the application. As in Chapter 1 (Section 1.3), let W_{base} denote the base execution time of an application without any overhead due to resilience techniques (without loss of generality, we assume unit-speed execution). Suppose the execution is divided into periodic patterns, defined by $\mathrm{P}(W, n, \boldsymbol{\alpha}, \boldsymbol{m}, \langle \boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_n \rangle)$. Let $\mathbb{E}(\mathrm{P})$ be the expected execution time of the pattern. For large jobs, the expected makespan W_{final} of the application when taking failures into account can then be approximated by:

$$W_{\text{final}} \approx \frac{\mathbb{E}(\mathrm{P})}{W} \times W_{\text{base}}.$$

Now, define $H(\mathrm{P}) = \frac{\mathbb{E}(\mathrm{P})}{W} - 1$ to be the expected *overhead* of the pattern. We obtain $W_{\text{final}} \approx W_{\text{base}} + H(\mathrm{P}) \times W_{\text{base}}$. Thus, minimizing the expected makespan is equivalent to minimizing the pattern overhead $H(\mathrm{P})$. Hence, we will focus on minimizing the pattern overhead in this chapter.

2.3 Revisiting Young and Daly

In this section, we revisit Young [97] and Daly [36] on computing the best periodic checkpointing interval, and extend their formula to include both fail-stop and silent errors. The result on the order of the optimal interval and the observations established in this case will pave the way for the subsequent analysis on more advanced patterns.

2.3.1 Optimal disk checkpointing interval

The classical formula by Young and Daly gives the optimal disk checkpointing interval without considering silent errors, thus does not include verification and memory checkpoints in the pattern. To cope with both fail-stop and silent errors, we analyze the pattern $\mathrm{P}_D = \mathrm{P}(W, 1, [1], [1], \langle [1] \rangle)$, which contains one single segment with a unique chunk followed by a guaranteed verification, a memory checkpoint and a disk checkpoint (see Figure 2.1).

Obviously, the only parameter to determine is the work length W, which is also referred to as the checkpointing period by Young/Daly [36, 97]. The following proposition shows the expected execution time of a pattern with a fixed work length.

Proposition 2. *The expected execution time of a given pattern* $\mathrm{P}(W, 1, [1], [1], \langle [1] \rangle)$ *is*

$$\begin{aligned} \mathbb{E}(\mathrm{P}) = W + V^* + C_M + C_D + \left(\lambda_s + \frac{\lambda_f}{2}\right) W^2 \\ + \lambda_s W (V^* + R_M) + \lambda_f W (R_M + R_D) + O(\lambda^2 W^3) \,. \end{aligned} \tag{2.1}$$

Proof. Let $p^f = 1 - e^{-\lambda_f W}$ and $p^s = 1 - e^{-\lambda_s W}$ denote the probabilities of having at least one fail-stop error and at least one silent error, respectively, in the pattern. The expected execution time can be expressed using the following recursive formula:

$$\begin{aligned} \mathbb{E}(\mathrm{P}) = p^f \left(\mathbb{E}(T^{\text{lost}}) + R_D + R_M + \mathbb{E}(\mathrm{P}) \right) \\ + (1 - p^f) \big(W + V^* + p^s (R_M + \mathbb{E}(\mathrm{P})) \\ + (1 - p^s)(C_M + C_D) \big) \,, \end{aligned} \tag{2.2}$$

where $\mathbb{E}(T^{\text{lost}})$ denotes the expected time loss during the execution of the pattern if a fail-stop error strikes. Equation (2.2) can be interpreted as follows: if a fail-stop error occurs, we lose $\mathbb{E}(T^{\text{lost}})$ time, perform a recovery from both disk and memory, and then re-execute the pattern (Line 1). If no fail-stop error strikes during the execution, we run the guaranteed verification to detect silent errors, which if indeed occurred involves a memory recovery only followed by a re-execution (Line 2). Otherwise, if no silent error strikes either, we can proceed with the memory and disk checkpointing (Line 3).

To derive the expected execution time, we need to compute $\mathbb{E}(T^{\text{lost}})$, which can be expressed as follows:

$$\begin{aligned} \mathbb{E}(T^{\text{lost}}) &= \int_0^\infty x \mathbb{P}(X = x | X < W) dx \\ &= \frac{1}{\mathbb{P}(X < W)} \int_0^W x \mathbb{P}(X = x) dx \,, \end{aligned}$$

where $\mathbb{P}(X = x)$ denotes the probability that a fail-stop error strikes at time x. By definition, we have $\mathbb{P}(X = x) = \lambda_f e^{-\lambda_f x}$ and $\mathbb{P}(X < W) = 1 - e^{-\lambda_f W}$. Integrating by parts, we get

$$\mathbb{E}(T^{\text{lost}}) = \frac{1}{\lambda_f} - \frac{W}{e^{\lambda_f W} - 1} \,. \tag{2.3}$$

Now, substituting Equation (2.3) into Equation (2.2) and simplifying, we obtain

$$\begin{aligned}\mathbb{E}(\mathrm{P}) = {} & \frac{e^{(\lambda_f+\lambda_s)W} - e^{\lambda_s W}}{\lambda_f} + e^{\lambda_s W} \cdot V^* + C_D + C_M \\ & + \left(e^{(\lambda_f+\lambda_s)W} - e^{\lambda_s W}\right) R_D + \left(e^{(\lambda_f+\lambda_s)W} - 1\right) R_M \,.\end{aligned}$$

By approximating $e^{\lambda x} = 1 + \lambda x + \frac{\lambda^2 x^2}{2}$ up to the second-order term, we can further simplify the expected execution time, which turns out to be given by Equation (2.1). □

Theorem 5. *A first-order approximation to the optimal work length in pattern* $\mathrm{P}(W, 1, [1], [1], \langle [1] \rangle)$ *is given by*

$$W^* = \sqrt{\frac{V^* + C_M + C_D}{\lambda_s + \frac{\lambda_f}{2}}} \,. \tag{2.4}$$

The optimal expected overhead is

$$H^*(\mathrm{P}) = 2\sqrt{\left(\lambda_s + \frac{\lambda_f}{2}\right)(V^* + C_M + C_D)} + O(\lambda) \,. \tag{2.5}$$

Proof. From the result of Proposition 2, the expected overhead of the pattern can be computed as

$$\begin{aligned}H(\mathrm{P}) = {} & \frac{V^* + C_M + C_D}{W} + \left(\lambda_s + \frac{\lambda_f}{2}\right) W \\ & + \lambda_s(V^* + R_M) + \lambda_f(R_M + R_D) + O(\lambda^2 W^2) \,.\end{aligned} \tag{2.6}$$

Assume that the platform MTBF $\mu = 1/\lambda$ is large in front of the resilience parameters. Then consider the first two terms of $H(\mathrm{P})$ (Line 1 of Equation (2.6)): the overhead is minimal when the pattern has length $W = \Theta(\lambda^{-1/2})$, and in that case both terms are of order $\Theta(\lambda^{1/2})$, so that we have

$$H(\mathrm{P}) = \Theta(\lambda^{1/2}) + O(\lambda).$$

Indeed, the last term $O(\lambda^2 W^2)$ becomes also negligible compared to $\Theta(\lambda^{1/2})$. Hence, the optimal pattern length W^* can be obtained by balancing the first two terms in the above expression, which gives Equation (2.4). Then, by substituting W^* back into $H(\mathrm{P})$, we get the optimal expected overhead as shown by Equation (2.5). □

Remarks. When only fail-stop errors exist, there is no need to perform verification and memory checkpointing: we retrieve the classical formula by Young [97] and Daly[36], which is given by $W^* = \sqrt{2C_D/\lambda_f}$. When there are only silent errors, we do not need to perform disk checkpointing, and the optimal work length is given by $W^* = \sqrt{(V^* + C_M)/\lambda_s}$.

2.3.2 Observations

First, we observe from Theorem 5 that the optimal work length W^* of a pattern is in the order of $\Theta\left(\lambda^{-1/2}\right)$ and the optimal overhead $H^*(\mathrm{P})$ is in the order of $\Theta(\lambda^{1/2})$. This allows us to express the expected execution overhead of a pattern in the following general form:

$$H(\mathrm{P}) = \frac{o_{\text{ef}}}{W} + o_{\text{rw}} W + O(\lambda) \ , \tag{2.7}$$

where o_{ef} and o_{rw} are two key parameters that characterize two different types of overheads in the execution, and they are defined below.

Definition 3. *For a given pattern, o_{ef} denotes the* error-free *overhead due to the resilience operations (e.g., verification, checkpointing), and o_{rw} denotes the* re-executed work *overhead, in terms of the fraction of re-executed work due to errors.*

In the simple pattern $\mathrm{P}(W, 1, [1], [1], \langle [1] \rangle)$ analyzed above, these two overheads are given by $o_{\text{ef}} = V^* + C_M + C_D$ and $o_{\text{rw}} = \lambda_s + \frac{\lambda_f}{2}$, respectively.

Therefore, from Equation (2.7), the optimal pattern length and the optimal expected overhead can be expressed as

$$W^* = \sqrt{\frac{o_{\text{ef}}}{o_{\text{rw}}}} \ , \tag{2.8}$$

$$H^*(\mathrm{P}) = 2\sqrt{o_{\text{ef}} \times o_{\text{rw}}} + O(\lambda) \ . \tag{2.9}$$

We can see that minimizing the expected execution overhead $H(\mathrm{P})$ of a pattern becomes equivalent to minimizing the product $o_{\text{ef}} \times o_{\text{rw}}$ up to the dominating term, which is coherent with Chapter 1 (Section 1.5.3). Intuitively, including more resilience operators reduces the re-executed work overhead but adversely increases the error-free overhead, and vice versa. This requires a resilience protocol that finds the optimal tradeoff between o_{ef} and o_{rw}. We will make use of this observation in the next section to derive the optimal patterns in more complicated protocols.

2.4 Optimal patterns

In this section, we derive the optimal pattern that involves two levels of checkpointing coupled with verifications. We start with simpler patterns that do not contain any intermediate verification nor memory checkpoint, and then move on to settle the complete full pattern.

2.4.1 Pattern $\mathbf{P}_{DM} = \mathbf{P}(W, n, \boldsymbol{\alpha}, [1, \ldots, 1], \langle [1], \ldots, [1] \rangle)$

We first consider a pattern that contains multiple segments, but each segment has only one chunk. In other words, the protocol performs multiple memory checkpoints between two disk checkpoints but without any intermediate verification. Figure 2.3 depicts the pattern P_{DM} in this protocol.

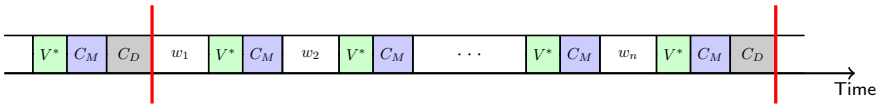


Figure 2.3: Pattern $\mathrm{P}_{DM} = \mathrm{P}(W, n, \boldsymbol{\alpha}, [1, \ldots, 1], \langle [1], \ldots, [1] \rangle)$.

The goal is to determine the pattern work length W, the number of memory checkpoints n, and the relative lengths of the segments $\boldsymbol{\alpha}$ inside the pattern. The following proposition shows the expected execution time of a pattern when these parameters are fixed.

Proposition 3. *The expected execution time of a given pattern* $\mathrm{P}(W, n, \boldsymbol{\alpha}, [1, \ldots, 1], \langle [1], \ldots, [1] \rangle)$ *is*

$$\begin{aligned} \mathbb{E}(\mathrm{P}) &= W + n(V^* + C_M) + C_D \\ &\quad + \left(\lambda_s \sum_{i=1}^{n} \alpha_i^2 + \frac{\lambda_f}{2} \right) W^2 + O(\sqrt{\lambda}) \,. \end{aligned} \tag{2.10}$$

Proof. Define E_i as the expected time to execute the i-th segment of the pattern up to the memory checkpoint at the end of the segment. We first show the following result on E_i:

$$E_i = w_i + V^* + C_M + \lambda_s w_i^2 + \lambda_f \left(\frac{w_i^2}{2} + \sum_{k=1}^{i-1} w_k w_i \right) + O(\sqrt{\lambda}) \,,$$

where $w_i = \alpha_i W$ denotes the work length of the i-th segment.

We prove the above claim by induction on i. For the base case, the problem is reduced to the simple pattern shown in Section 2.3.1, except that there is no disk checkpoint. Since we know from Theorem 5 that the work length of a pattern is in the order of $\Theta\left(\lambda^{-1/2}\right)$, we get the following result from Proposition 2:

$$E_1 = w_1 + V^* + C_M + \lambda_s w_1^2 + \frac{\lambda_f}{2} w_1^2 + O(\sqrt{\lambda}) \,.$$

Suppose the claim holds up to E_{i-1}. Then, E_i can be expressed recursively as follows:

$$\begin{aligned} E_i = p_i^f &\left(\mathbb{E}(T_i^{\text{lost}}) + R_D + R_M + \sum_{k=1}^{i-1} E_k + E_i \right) \\ &+ (1 - p_i^f)\big(w_i + V^* + p_i^s (R_M + E_i) + (1 - p_i^s) C_M\big) \,, \end{aligned}$$

where $\mathbb{E}(T_i^{\text{lost}})$ denotes the expected time loss during the execution of segment i when a fail-stop error strikes, which according to Equation (2.3) is given by $\mathbb{E}(T_i^{\text{lost}}) = \frac{1}{\lambda_f} - \frac{w_i}{e^{\lambda_f w_i} - 1}$, and p_i^f and p_i^s denote the probabilities of having at least one fail-stop error and at least one silent error in segment i, respectively. By following the reasoning of the proof of Proposition 2, we

obtain:

$$\begin{aligned} E_i &= w_i + V^* + C_M + \lambda_s w_i^2 + \frac{\lambda_f}{2} w_i^2 + \lambda_f w_i \sum_{k=1}^{i-1} E_k + O(\sqrt{\lambda}) \\ &= w_i + V^* + C_M + \lambda_s w_i^2 + \frac{\lambda_f}{2} w_i^2 + \lambda_f w_i \sum_{k=1}^{i-1} (w_k + O(1)) + O(\sqrt{\lambda}) \\ &= w_i + V^* + C_M + \lambda_s w_i^2 + \lambda_f \left(\frac{w_i^2}{2} + (\sum_{k=1}^{i-1} w_k) w_i \right) + O(\sqrt{\lambda}) \, . \end{aligned}$$

Now, we compute the expected execution time of the pattern by summing up all the E_i's as follows:

$$\begin{aligned} \mathbb{E}(\mathrm{P}) &= \sum_{i=1}^{n} E_i + C_D \\ &= \sum_{i=1}^{n} w_i + n(V^* + C_M) + C_D \\ &\quad + \lambda_s \sum_{i=1}^{n} w_i^2 + \lambda_f \sum_{i=1}^{n} \left(\frac{w_i^2}{2} + (\sum_{k=1}^{i-1} w_k) w_i \right) + O(\sqrt{\lambda}) \\ &= W + n(V^* + C_M) + C_D \\ &\quad + \left(\lambda_s \sum_{i=1}^{n} \alpha_i^2 + \frac{\lambda_f}{2} \right) W^2 + O(\sqrt{\lambda}) \, , \end{aligned}$$

since $\sum_{i=1}^{n} \left(w_i^2 + 2(\sum_{k=1}^{i-1} w_k) w_i \right) = (\sum_{i=1}^{n} w_i)^2 = W^2$. □

Theorem 6. *A first-order approximation to the optimal parameters in pattern* $\mathrm{P}(W, n, \boldsymbol{\alpha}, [1, \ldots, 1], \langle [1], \ldots, [1] \rangle)$ *is given by*

$$\alpha_i^* = \frac{1}{n^*} \textit{ for } 1 \leq i \leq n^* \, , \tag{2.11}$$

$$W^* = \sqrt{\frac{n^*(V^* + C_M) + C_D}{\frac{\lambda_s}{n^*} + \frac{\lambda_f}{2}}} \, , \tag{2.12}$$

and n^* *is either* $\max(1, \lfloor \bar{n}^* \rfloor)$ *or* $\lceil \bar{n}^* \rceil$*, where*

$$\bar{n}^* = \sqrt{\frac{2\lambda_s}{\lambda_f} \cdot \frac{C_D}{V^* + C_M}} \, . \tag{2.13}$$

The optimal expected overhead is

$$H^*(\mathrm{P}) = 2\sqrt{\lambda_s (V^* + C_M)} + \sqrt{2\lambda_f C_D} + O(\lambda) \, . \tag{2.14}$$

Proof. Given the number of segments n and subject to $\sum_{i=1}^{n} \alpha_i = 1$, we know that $\sum_{i=1}^{n} \alpha_i^2$ is minimized when $\alpha_i = \frac{1}{n}$ for all $1 \leq i \leq n$. Hence, we can derive the two types of overheads from Proposition 3 as follows:

$$o_{\text{ef}} = n(V^* + C_M) + C_D \ ,$$
$$o_{\text{rw}} = \frac{\lambda_s}{n} + \frac{\lambda_f}{2} \ .$$

For a given n, the optimal work length $W^* = \sqrt{\frac{o_{\text{ef}}}{o_{\text{rw}}}}$ is therefore given by Equation (2.12). Now, minimizing $F(n) = o_{\text{ef}} \times o_{\text{rw}} = (n(V^* + C_M) + C_D)\left(\frac{\lambda_s}{n} + \frac{\lambda_f}{2}\right)$, we get the optimal value of $\bar{n}^*$ as shown in Equation (2.13). Since the number of segments can only be a positive integer, and $F(n)$ is a convex function of n, the optimal integer solution is either $\max(1, \lfloor \bar{n}^* \rfloor)$ or $\lceil \bar{n}^* \rceil$, whichever one leads to a smaller value of $F(n)$. Substituting Equation (2.13) back into $H^*(\text{P}) = 2\sqrt{o_{\text{ef}} \times o_{\text{rw}}}$, we obtain the optimal expected overhead as shown in Equation (2.14). □

Remarks. We can see why the analysis conducted here is different from multi-level checkpointing with two levels of fail-stop errors. In the latter case, one has to make a case study depending on which error type strikes first, while in our case, silent errors do not interrupt the execution and are detected at the end of the segments.

2.4.2 Pattern $\text{P}_{DV} = \text{P}(W, 1, [1], [m], \langle \boldsymbol{\beta} \rangle)$

We now consider a pattern that contains only one segment, which has multiple chunks in it. Each chunk ends with a partial verification, except the last one, which ends with a guaranteed verification followed by a memory checkpoint and a disk checkpoint. Figure 2.4 depicts the pattern P_{DV} in this protocol.

For simplicity, let m (instead of m_1) denote the number of chunks in the pattern, and let w_j (instead of $w_{1,j}$) denote the length of the j-th chunk for $1 \leq j \leq m$. We define $\beta_j = w_j/W$. The goal is to determine the pattern work length W, the number of chunks m as well as their relative lengths $\boldsymbol{\beta}$.

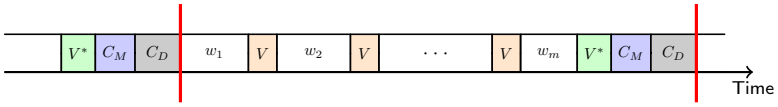


Figure 2.4: Pattern $\text{P}_{DV} = \text{P}(W, 1, [1], [m], \langle \boldsymbol{\beta} \rangle)$.

Proposition 4. *The expected execution time of a given pattern* $\text{P}(W, 1, [1], [m], \langle \boldsymbol{\beta} \rangle)$ *is*

$$\mathbb{E}(\text{P}) = W + (m-1)V + V^* + C_M + C_D + \left(\lambda_s \boldsymbol{\beta}^T A \boldsymbol{\beta} + \frac{\lambda_f}{2}\right) W^2 + O(\sqrt{\lambda}) \ , \tag{2.15}$$

where A is an $m \times m$ symmetric matrix defined by

$$A_{i,j} = \frac{1}{2}\left(1 + (1-r)^{|i-j|}\right) , \tag{2.16}$$

for all $1 \leq i, j \leq m$.

Proof. We first define some notations to be used in the proof. Let $p_j^f = 1 - e^{-\lambda_f w_j}$ and $p_j^s = 1 - e^{-\lambda_s w_j}$ denote the probabilities of having at least one fail-stop error and at least one silent error in chunk j, respectively. Let V_j denote the cost of the verification right after chunk j, so we have $V_j = V$ for $1 \leq j \leq m-1$ and $V_m = V^*$. Finally, let $\mathbb{E}(T_j^{\text{lost}})$ denote the expected time loss during the execution of chunk j if a fail-stop error strikes in this chunk. Based on Equation (2.3), we have $\mathbb{E}(T_j^{\text{lost}}) = \frac{1}{\lambda_f} - \frac{w_j}{e^{\lambda_f w_j}-1}$.

To derive the expected execution time of the pattern, we need to know the probability that chunk j actually gets executed in the current attempt. Let q_j denote this probability; we compute it as follows. The first chunk is always executed, so we have $q_1 = 1$. Consider the second chunk, which is executed when there is no fail-stop error and no silent error in the first chunk. However, for silent errors that did occur in the first chunk, the partial verification V_1 may have missed them with probability $1-r$. In this case, the second chunk also gets executed. Hence, we have $q_2 = (1-p_1^f)\big((1-p_1^s) + p_1^s(1-r)\big)$. In general, the probability that the j-th chunk gets executed can be written as:

$$q_j = \left(\prod_{k=1}^{j-1}(1-p_k^f)\right)\left(\prod_{k=1}^{j-1}(1-p_k^s) + g_j\right) ,$$

where g_j denotes the probability that silent errors actually occurred before chunk j, but have been missed by all the partial verifications up to V_{j-1}, thus enabling chunk j to be executed. By enumerating all possible locations where silent errors could strike, we can express g_j as:

$$g_j = \sum_{\ell=1}^{j-1}\left(\prod_{k=1}^{\ell-1}(1-p_k^s)\right) p_\ell^s (1-r)^{j-\ell} .$$

Now, we are ready to compute the expected execution time of the pattern. The following gives the recursive expression:

$$\begin{aligned}\mathbb{E}(\mathbf{P}) = &\left(\prod_{k=1}^{m}(1-p_k^f)(1-p_k^s)\right)(C_M + C_D) \\ &+ \left(1 - \prod_{k=1}^{m}(1-p_k^f)(1-p_k^s)\right)(R_M + \mathbb{E}(\mathbf{P})) \\ &+ \sum_{j=1}^{m} q_j \left(p_j^f\left(\mathbb{E}(T_j^{\text{lost}}) + R_D\right) + (1-p_j^f)(w_j + V_j)\right) .\end{aligned} \tag{2.17}$$

Specifically, Line 1 of Equation (2.17) shows that the memory and disk checkpoints at the end of the pattern are performed only when neither fail-stop nor silent error has occurred in

all chunks. Under all the other cases, we need to re-execute the pattern as shown in Line 2. Regardless of the type of error that triggered the re-execution, we always need to restore the memory checkpoint. Finally, Line 3 shows the condition for each chunk j to be executed. The execution of the chunk is either completed or interrupted by a fail-stop error, in which case we lose $\mathbb{E}(T_j^{\text{lost}})$ time and need to additionally restore the disk checkpoint.

By simplifying Equation (2.17) and approximating the expression up to the second-order term, as in the proofs of Propositions 2 and 3, we obtain

$$\begin{aligned}\mathbb{E}(\mathrm{P}) &= W + (m-1)V + V^* + C_M + C_D \\ &\quad + \lambda_s f W^2 + \frac{\lambda_f}{2} W^2 + O(\sqrt{\lambda}) \,,\end{aligned}$$

where $f = \sum_{j=1}^{m} \beta_j \left(\sum_{k=1}^{j-1} \beta_k (1-r)^{j-k} + \sum_{k=j}^{m} \beta_k \right)$, and it can be concisely written as $f = \boldsymbol{\beta}^T M \boldsymbol{\beta}$, where M is the $m \times m$ matrix given by

$$M_{i,j} = \begin{cases} 1 & \text{for } i \leq j \\ (1-r)^{i-j} & \text{for } i > j \end{cases} .$$

By replacing M by $A = \frac{M+M^T}{2}$, which does not affect the value of f, we obtain the symmetric matrix A in Equation (2.16) and the expected execution time in Equation (2.15). □

Theorem 7. *A first-order approximation to the optimal parameters in pattern* $\mathrm{P}(W, 1, [1], [m], \langle \boldsymbol{\beta} \rangle)$ *is given by*

$$\beta_j^* = \begin{cases} \frac{1}{(m^*-2)r+2} & \text{for } j = 1, m^* \\ \frac{r}{(m^*-2)r+2} & \text{for } 2 \leq j \leq m^* - 1 \end{cases} , \tag{2.18}$$

$$W^* = \sqrt{\frac{(m^*-1)V + V^* + C_M + C_D}{\frac{1}{2}\left(1 + \frac{2-r}{(m^*-2)r+2}\right)\lambda_s + \frac{\lambda_f}{2}}} , \tag{2.19}$$

and m^* *is either* $\max(1, \lfloor \bar{m}^* \rfloor)$ *or* $\lceil \bar{m}^* \rceil$*, where*

$$\bar{m}^* = 2 - \frac{2}{r} + \sqrt{\frac{\lambda_s}{\lambda_s + \lambda_f} \frac{2-r}{r} \left(\frac{V^* + C_M + C_D}{V} - \frac{2-r}{r} \right)} . \tag{2.20}$$

The optimal expected overhead is

$$\begin{aligned} H^*(\mathrm{P}) &= \sqrt{2(\lambda_s + \lambda_f)\left(V^* - \frac{2-r}{r}V + C_M + C_D\right)} \\ &\quad + \sqrt{2\lambda_s \frac{2-r}{r} V} + O(\lambda) \,. \end{aligned} \tag{2.21}$$

Proof. Given the number of chunks m and subject to $\sum_{j=1}^{m} \beta_j = 1$, it has been shown in Chapter 1 (Section 1.5.4) that function $f = \boldsymbol{\beta}^T A \boldsymbol{\beta}$ is minimized when $\boldsymbol{\beta}$ follows Equation

(2.18), and its minimum value is given by $f^* = \frac{1}{2}\left(1 + \frac{2-r}{(m-2)r+2}\right)$. From Proposition 3, we can derive the two types of overheads as follows:

$$o_{\text{ef}} = (m-1)V + V^* + C_M + C_D \ ,$$
$$o_{\text{rw}} = \frac{1}{2}\left(1 + \frac{2-r}{(m-2)r+2}\right)\lambda_s + \frac{\lambda_f}{2} \ .$$

The optimal work length $W^* = \sqrt{\frac{o_{\text{ef}}}{o_{\text{rw}}}}$ for any fixed m is thus given by Equation (2.19). The optimal number of chunks $\bar{m}^*$ shown in Equation (2.20) is obtained by minimizing $F(m) = o_{\text{ef}} \times o_{\text{rw}} = \frac{1}{2}\big((m-1)V + V^* + C_M + C_D\big)\left(\left(1 + \frac{2-r}{(m-2)r+2}\right)\lambda_s + \lambda_f\right)$. Again, the number of chunks in a pattern can only be a positive integer, so m^* is either $\max(1, \lfloor \bar{m}^* \rfloor)$ or $\lceil \bar{m}^* \rceil$, since $F(m)$ is a convex function of m. Finally, substituting Equation (2.20) back into $H^*(\text{P}) = 2\sqrt{o_{\text{ef}} \times o_{\text{rw}}}$ gives rise to the optimal expected overhead as shown in Equation (2.21). □

Remarks. When only guaranteed verification is used, the optimal pattern contains equal-length chunks. In this case, the pattern is denoted P_{DV^*}, and we have:

$$\beta_j^* = \frac{1}{m^*} \text{ for } 1 \le j \le m^* \ ,$$
$$W^* = \sqrt{\frac{m^*V^* + C_M + C_D}{\frac{1}{2}\left(1 + \frac{1}{m^*}\right)\lambda_s + \frac{\lambda_f}{2}}} \ ,$$
$$\bar{m}^* = \sqrt{\frac{\lambda_s}{\lambda_s + \lambda_f} \cdot \frac{C_M + C_D}{V^*}} \ ,$$
$$H^*(\text{P}) = \sqrt{2(\lambda_s + \lambda_f)(C_M + C_D)} + \sqrt{2\lambda_s V} + O(\lambda) \ .$$

2.4.3 Pattern $\text{P}_{DMV} = \text{P}(W, n, \boldsymbol{\alpha}, \boldsymbol{m}, \langle \boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_n \rangle)$

Finally, we consider the complete pattern that contains multiple segments, each of which has multiple chunks. This represents the general two-level checkpointing protocol with intermediate verifications for silent error detection. Figure 2.4 depicts the pattern in this protocol.

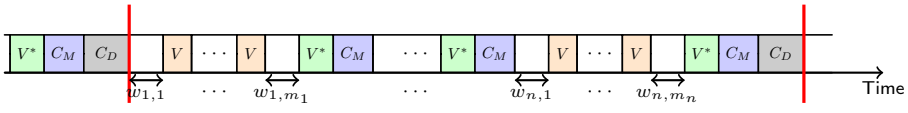


Figure 2.5: Pattern $\text{P}_{DMV} = \text{P}(W, n, \boldsymbol{\alpha}, \boldsymbol{m}, \langle \boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_n \rangle)$.

The goal is to determine all the parameters of the pattern. Again, we first derive the expected execution time of a pattern when all parameters are fixed.

Proposition 5. *The expected execution time of a given pattern* $\mathrm{P}(W, n, \boldsymbol{\alpha}, \boldsymbol{m}, \langle \boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_n \rangle)$ *is*

$$\begin{aligned}\mathbb{E}(\mathrm{P}) = W &+ \sum_{i=1}^{n} (m_i - 1)V + n(V^* + C_M) + C_D \\ &+ \left(\lambda_s \sum_{i=1}^{n} \boldsymbol{\beta}_i^T A^{(m_i)} \boldsymbol{\beta}_i \cdot \alpha_i^2 + \frac{\lambda_f}{2} \right) W^2 + O(\sqrt{\lambda}) \,, \end{aligned} \tag{2.22}$$

where $A^{(m)}$ *denotes an* $m \times m$ *symmetric matrix*[1] *defined by* $A^{(m)}_{i,j} = \frac{1}{2}\left(1 + (1-r)^{|i-j|}\right)$ *for all* $1 \leq i, j \leq m$.

Proof. Define E_i to be the expected execution time of the i-th segment up to the memory checkpoint at the end of the segment. We first show the following result:

$$\begin{aligned} E_i = w_i &+ (m_i - 1)V + V^* + C_M + \lambda_s \boldsymbol{\beta}_i^T A^{(m_i)} \boldsymbol{\beta}_i \cdot w_i^2 \\ &+ \lambda_f \left(\frac{w_i^2}{2} + \sum_{k=1}^{i-1} w_k w_i \right) + O(\sqrt{\lambda}) \,. \end{aligned}$$

The proof combines the techniques from those of Propositions 3 and 4. Specifically, as in the proof of Proposition 3, we go by induction on i. The base case is equivalent to the pattern $\mathrm{P}(W, 1, [1], [m], \langle \boldsymbol{\beta} \rangle)$ analyzed in Section 2.4.2, except that there is no disk checkpoint at the end of the segment. Hence, from Proposition 4, we get

$$\begin{aligned} E_1 = w_1 &+ (m_1 - 1)V + V^* + C_M \\ &+ \lambda_s \boldsymbol{\beta}_1^T A^{(m_1)} \boldsymbol{\beta}_1 \cdot w_1^2 + \frac{\lambda_f}{2} w_1^2 + O(\sqrt{\lambda}) \,. \end{aligned}$$

Suppose the claim holds up to E_{i-1}. Then, by following the proof of Proposition 4, in particular, Equation (2.17), we can express E_i recursively as follows:

$$\begin{aligned} E_i = &\left(\prod_{j=1}^{m_i} (1 - p^f_{i,j})(1 - p^s_{i,j}) \right) C_M \\ &+ \left(1 - \prod_{j=1}^{m_i} (1 - p^f_{i,j})(1 - p^s_{i,j}) \right) (R_M + E_i) \\ &+ \sum_{j=1}^{m_i} q_{i,j} \left(p^f_{i,j} \left(\mathbb{E}(T^{\text{lost}}_{i,j}) + R_D + \sum_{k=1}^{i-1} E_k \right) + (1 - p^f_{i,j})(w_{i,j} + V_{i,j}) \right), \end{aligned} \tag{2.23}$$

where $q_{i,j}$ denotes the probability that the j-th chunk of the i-th segment gets executed, and it is given by

$$q_{i,j} = \left(\prod_{k=1}^{j-1} (1 - p^f_{i,k}) \right) \left(\prod_{k=1}^{j-1} (1 - p^s_{i,k}) + g_{i,j} \right) \,,$$

[1] Matrices $A^{(m)}$ only differ in their dimensions; they have the same components.

with

$$g_{i,j} = \sum_{\ell=1}^{j-1} \left(\prod_{k=1}^{\ell-1} (1 - p_{i,k}^s) \right) p_{i,\ell}^s (1-r)^{j-\ell} .$$

We point out two differences between Equations (2.17) and (2.23). First, we do not need to perform a disk checkpoint when there is no error in segment i (Line 1). Second, if a fail-stop error occurred in the j-th chunk, we need to additionally re-execute all the segments before i (Line 3). Simplifying and approximating Equation (2.23) as in the proof of Proposition 4, we get:

$$\begin{aligned}
E_i &= w_i + (m_i - 1)V + V^* + C_M + \lambda_s \boldsymbol{\beta}_i^T A^{(m_i)} \boldsymbol{\beta}_i \cdot w_i^2 \\
&\quad + \frac{\lambda_f}{2} w_i^2 + \lambda_f w_i \sum_{k=1}^{i-1} E_k + O(\sqrt{\lambda}) \\
&= w_i + (m_i - 1)V + V^* + C_M + \lambda_s \boldsymbol{\beta}_i^T A^{(m_i)} \boldsymbol{\beta}_i \cdot w_i^2 \\
&\quad + \frac{\lambda_f}{2} w_i^2 + \lambda_f w_i \sum_{k=1}^{i-1} (w_k + O(1)) + O(\sqrt{\lambda}) \\
&= w_i + (m_i - 1)V + V^* + C_M + \lambda_s \boldsymbol{\beta}_i^T A^{(m_i)} \boldsymbol{\beta}_i \cdot w_i^2 \\
&\quad + \lambda_f \left(\frac{w_i^2}{2} + \sum_{k=1}^{i-1} w_k w_i \right) + O(\sqrt{\lambda}) .
\end{aligned}$$

Now, we can compute the expected execution time of the pattern by summing up all the E_i's as follows:

$$\begin{aligned}
\mathbb{E}(\mathrm{P}) &= \sum_{i=1}^{n} E_i + C_D \\
&= \sum_{i=1}^{n} w_i + \sum_{i=1}^{n} (m_i - 1)V + n(V^* + C_M) + C_D \\
&\quad + \lambda_s \sum_{i=1}^{n} \boldsymbol{\beta}_i^T A^{(m_i)} \boldsymbol{\beta}_i \cdot w_i^2 + \lambda_f \sum_{i=1}^{n} \left(\frac{w_i^2}{2} + \sum_{k=1}^{i-1} w_k w_i \right) + O(\sqrt{\lambda}) \\
&= W + \sum_{i=1}^{n} (m_i - 1)V + n(V^* + C_M) + C_D \\
&\quad + \left(\lambda_s \sum_{i=1}^{n} \boldsymbol{\beta}_i^T A^{(m_i)} \boldsymbol{\beta}_i \cdot \alpha_i^2 + \frac{\lambda_f}{2} \right) W^2 + O(\sqrt{\lambda}) .
\end{aligned}$$

This completes the proof of the proposition. □

Theorem 8. *A first-order approximation to the optimal parameters in pattern* $\mathrm{P}(W, n, \boldsymbol{\alpha}, \boldsymbol{m}, \langle \boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_n \rangle)$ *is given by*

$$\alpha_i^* = \frac{1}{n^*} \text{ for } i = 1 \ldots n^* , \tag{2.24}$$

$$\beta_{i,j}^* = \begin{cases} \frac{1}{(m_i^*-2)r+2} & \text{for } 1 \le i \le n^*, j = 1, m_i^* \\ \frac{r}{(m_i^*-2)r+2} & \text{for } 1 \le i \le n^*, 2 \le j \le m_i^* - 1 \end{cases} , \tag{2.25}$$

$$W^* = \sqrt{\frac{n^*(m^*-1)V + n^*(V^* + C_M) + C_D}{\frac{1}{2}\left(1 + \frac{2-r}{(m^*-2)r+2}\right)\frac{\lambda_s}{n^*} + \frac{\lambda_f}{2}}} , \tag{2.26}$$

and n^* *is either* $\max(1, \lfloor \bar{n}^* \rfloor)$ *or* $\lceil \bar{n}^* \rceil$*, and* m_i^* *is either* $\max(1, \lfloor \bar{m}^* \rfloor)$ *or* $\lceil \bar{m}^* \rceil$ *for all* $1 \le i \le n^*$*, where*

$$\bar{n}^* = \sqrt{\frac{\lambda_s}{\lambda_f} \cdot \frac{C_D}{V^* - \frac{2-r}{r}V + C_M}} , \tag{2.27}$$

$$\bar{m}^* = 2 - \frac{2}{r} + \sqrt{\frac{2-r}{r}\left(\frac{V^* + C_M}{V} - \frac{2-r}{r}\right)} . \tag{2.28}$$

The optimal expected overhead is

$$H^*(\mathrm{P}) = \sqrt{2\lambda_f C_D} + \sqrt{2\lambda_s\left(V^* - \frac{2-r}{r}V + C_M\right)} + \sqrt{2\lambda_s \frac{2-r}{r} V} + O(\lambda) . \tag{2.29}$$

Proof. For any given n and $\boldsymbol{m}$, we perform a series of optimizations on the expected execution time shown in Equation (2.22). First, minimizing function $f_i = \boldsymbol{\beta}_i^T A^{(m_i)} \boldsymbol{\beta}_i$ subject to $\sum_{j=1}^{m_i} \beta_{i,j} = 1$ (as in the proof of Theorem 7), we get $f_i^* = \frac{1}{2}\left(1 + \frac{2-r}{(m_i-2)r+2}\right)$, obtained when $\boldsymbol{\beta}_i^*$ satisfies Equation (2.25). Next, minimizing $h = \sum_{i=1}^n f_i^* \alpha_i^2$ subject to $\sum_{i=1}^n \alpha_i = 1$, we get $h^* = \frac{1}{\sum_{i=1}^n 1/f_i^*}$, which is obtained at $\alpha_i^* = \frac{1/f_i^*}{\sum_{k=1}^n 1/f_k^*}$. Finally, subject to $\sum_{i=1}^n m_i = nm$, where m is the average number of chunks per segment, $\sum_{i=1}^n 1/f_i^*$ is maximized when $m_i = m$ for all $1 \le i \le n$. This means that $\boldsymbol{\alpha}^*$ satisfies Equation (2.24) and the minimum value of h^* is given by $h^* = \frac{1}{2n}\left(1 + \frac{2-r}{(m-2)r+2}\right)$.

Hence, we can write the two types of overheads as follows:

$$o_{\text{ef}} = n(m-1)V + n(V^* + C_M) + C_D ,$$
$$o_{\text{rw}} = \frac{1}{2}\left(1 + \frac{2-r}{(m-2)r+2}\right)\frac{\lambda_s}{n} + \frac{\lambda_f}{2} .$$

The optimal work length $W^* = \sqrt{\frac{o_{\text{ef}}}{o_{\text{rw}}}}$ for any fixed n and $\boldsymbol{m}$ is thus given by Equation (2.26).

Now, we need to find values for n and m that minimize the function $F(n,m) = o_{\text{ef}} \times o_{\text{rw}} = \frac{1}{2}\big(n(m-1)V + n(V^* + C_M) + C_D\big)\left(\left(1 + \frac{2-r}{(m-2)r+2}\right)\frac{\lambda_s}{n} + \lambda_f\right)$. For the solution $(\bar{n}^*, \bar{m}^*)$ given in Equations (2.27) and (2.28), we can verify that

$$\frac{\partial F(\bar{n}^*, \bar{m}^*)}{\partial n} = 0 \,,$$
$$\frac{\partial F(\bar{n}^*, \bar{m}^*)}{\partial m} = 0 \,,$$

and moreover

$$\frac{\partial^2 F(\bar{n}^*, \bar{m}^*)}{\partial n^2} > 0 \,,$$
$$\frac{\partial^2 F(\bar{n}^*, \bar{m}^*)}{\partial n^2} \cdot \frac{\partial^2 F(\bar{n}^*, \bar{m}^*)}{\partial n^2} - \left(\frac{\partial^2 F(\bar{n}^*, \bar{m}^*)}{\partial n \partial m}\right)^2 > 0 \,,$$

which shows that $(\bar{n}^*, \bar{m}^*)$ is indeed a global minimum of $F(n,m)$. Since the number of segments and number of chunks per segment can only be positive integers, and $F(n,m)$ is a convex function, the optimal integer solution is one of the four integer combinations around the optimal rational solution.

Finally, substituting Equations (2.27) and (2.28) into $H^*(\text{P}) = 2\sqrt{o_{\text{ef}} \times o_{\text{rw}}}$ and simplifying, we get the optimal expected overhead as shown in Equation (2.29). □

Remarks Theorem 8 shows that the optimal pattern has identical segments (same size and identical number and sizes of chunks). However, inside each segment, chunks do not necessarily have the same size. With partial verifications, the first and last chunk in each segment are larger than the other ones.

When only guaranteed verifications are used, all chunks in a segment (and hence in the whole pattern) have the same length. In this case, the pattern is denoted P_{DMV^*} and we have:

$$\alpha_i^* = \frac{1}{n^*} \text{ for } 1 \le i \le n^* \,,$$
$$\beta_{i,j}^* = \frac{1}{m^*} \text{ for } 1 \le i \le n^*, 1 \le j \le m^* \,,$$
$$W^* = \sqrt{\frac{n^* m^* V^* + n^* C_M + C_D}{\frac{1}{2}\left(1 + \frac{1}{m^*}\right)\frac{\lambda_s}{n^*} + \frac{\lambda_f}{2}}} \,,$$
$$\bar{n}^* = \sqrt{\frac{\lambda_s}{\lambda_f}\frac{C_D}{C_M}} \,,$$
$$\bar{m}^* = \sqrt{\frac{C_M}{V^*}} \,,$$
$$H^*(\text{P}) = \sqrt{2\lambda_f C_D} + \sqrt{2\lambda_s C_M} + \sqrt{2\lambda_s V^*} + O(\lambda) \,.$$

2.4.4 Summary of results

Table I summarizes the results. P_D, P_{DV^*} and P_{DV} are patterns with only one level of checkpointing, i.e., we always perform the memory checkpoint just before the disk checkpoint. P_{DV^*} adds extra guaranteed verifications between two disk checkpoints, while P_{DV} adds partial verifications. Similarly, P_{DM}, P_{DMV^*} and P_{DMV} correspond to two-levels checkpointing patterns, with extra guaranteed verifications for P_{DMV^*} and partial verifications for P_{DMV}.

We report in each case the optimal pattern length W^*, the optimal overhead $H(\mathrm{P})$, the optimal number of memory checkpoints n^* for the two-level checkpointing patterns, and the optimal number of verifications $\boldsymbol{m}^*$ within a segment when additional verifications are added.

Table I

THE SIX OPTIMAL PATTERNS. P_D, P_{DV^*} AND P_{DV} HAVE ONLY ONE LEVEL OF CHECKPOINTING, WHILE P_{DM}, P_{DMV^*} AND P_{DMV} HAVE TWO LEVELS. THE TABLE REPORTS THE OPTIMAL PATTERN LENGTH W^*, THE OPTIMAL OVERHEAD $H^*(\mathrm{P})$ (IGNORING LOWER-ORDER TERMS), THE OPTIMAL NUMBER OF MEMORY CHECKPOINTS n^* FOR THE TWO-LEVEL CHECKPOINTING PATTERNS, AND THE OPTIMAL NUMBER OF VERIFICATIONS $\boldsymbol{m}^*$ WITHIN A SEGMENT WHEN ADDITIONAL VERIFICATIONS ARE ADDED.

Pattern	W^*	n^*	$\boldsymbol{m}^*$	$H^*(\mathrm{P})$
P_D	$\sqrt{\frac{V^*+C_M+C_D}{\lambda_s+\frac{\lambda_f}{2}}}$	–	–	$2\sqrt{\left(\lambda_s+\frac{\lambda_f}{2}\right)(V^*+C_M+C_D)}$
P_{DV^*}	$\sqrt{\frac{m^*V^*+C_M+C_D}{\frac{1}{2}(1+\frac{1}{m^*})\lambda_s+\frac{\lambda_f}{2}}}$	–	$\sqrt{\frac{\lambda_s}{\lambda_s+\lambda_f}\cdot\frac{C_M+C_D}{V^*}}$	$\sqrt{2(\lambda_s+\lambda_f)C_M+C_D}+\sqrt{2\lambda_s V^*}$
P_{DV}	$\sqrt{\frac{(m^*-1)V+V^*+C_M+C_D}{\frac{1}{2}\left(1+\frac{2-r}{(m^*-2)r+2}\right)\lambda_s+\frac{\lambda_f}{2}}}$	–	$2-\frac{2}{r}+\sqrt{\frac{\lambda_s}{\lambda_s+\lambda_f}}$ $\times\sqrt{\frac{2-r}{r}\left(\frac{V^*+C_M+C_D}{V}-\frac{2-r}{r}\right)}$	$\sqrt{2(\lambda_s+\lambda_f)\left(V^*-\frac{2-r}{r}V+C_M+C_D\right)}$ $+\sqrt{2\lambda_s\frac{2-r}{r}V}$
P_{DM}	$\sqrt{\frac{n^*(V^*+C_M)+C_D}{\frac{\lambda_s}{n^*}+\frac{\lambda_f}{2}}}$	$\sqrt{\frac{2\lambda_s}{\lambda_f}\cdot\frac{C_D}{V^*+C_M}}$	–	$2\sqrt{\lambda_s(V^*+C_M)}+\sqrt{2\lambda_f C_D}$
P_{DMV^*}	$\sqrt{\frac{n^*m^*V^*+n^*C_M+C_D}{\frac{1}{2}(1+\frac{1}{m^*})\frac{\lambda_s}{n^*}+\frac{\lambda_f}{2}}}$	$\sqrt{\frac{\lambda_s}{\lambda_f}\cdot\frac{C_D}{C_M}}$	$\sqrt{\frac{C_M}{V^*}}$	$\sqrt{2\lambda_f C_D}+\sqrt{2\lambda_s C_M}+\sqrt{2\lambda_s V^*}$
P_{DMV}	$\sqrt{\frac{n^*(m^*-1)V+n^*(V^*+C_M)+C_D}{\frac{1}{2}\left(1+\frac{2-r}{(m^*-2)r+2}\right)\frac{\lambda_s}{n^*}+\frac{\lambda_f}{2}}}$	$\sqrt{\frac{\lambda_s}{\lambda_f}\cdot\frac{C_D}{V^*-\frac{2-r}{r}V+C_M}}$	$2-\frac{2}{r}$ $+\sqrt{\frac{2-r}{r}\left(\frac{V^*+C_M}{V}-\frac{2-r}{r}\right)}$	$\sqrt{2\lambda_f C_D}+\sqrt{2\lambda_s\left(V^*-\frac{2-r}{r}V+C_M\right)}$ $+\sqrt{2\lambda_s\frac{2-r}{r}V}$

2.5 Errors in verifications, checkpoints and recoveries

So far, we have assumed error-free execution during verifications, checkpoints and recoveries. In this section, we show how to handle fail-stop errors during these operations[2], and that the first-order approximations derived in the preceding section remain valid as long as the platform MTBF $\mu = 1/\lambda$ is large in front of the resilience parameters.

First, we handle errors during checkpoints and recoveries. The probability of experiencing at least one error during a (checkpoint or recovery) process of length L is given by $p_L^f = 1 - e^{-\lambda_f L}$ and, according to Equation (2.3), the expected time loss in executing this process is given by $\mathbb{E}(T_L^{\text{lost}}) = \frac{1}{\lambda_f} - \frac{L}{e^{\lambda_f L}-1}$. Let $\mathbb{E}(C_D)$, $\mathbb{E}(C_M)$, $\mathbb{E}(R_D)$ and $\mathbb{E}(R_M)$ denote the expected time

[2]Checkpoints and recoveries do not suffer from silent errors, since silent errors typically do not strike I/O and protected memory space, where memory checkpoints are assumed to be stored. Verifications can be protected from silent errors by using redundancy techniques to ensure correct results.

to perform disk checkpointing, memory checkpointing, disk recovery and memory recovery, respectively. Since each disk checkpoint is always preceded by a memory checkpoint, and each disk recovery is immediately followed by a memory recovery, we can express these expected execution times recursively as follows:

$$\mathbb{E}(R_D) = p^f_{R_D}\Big(\mathbb{E}(T^{\text{lost}}_{R_D}) + \mathbb{E}(R_D)\Big) + \Big(1 - p^f_{R_D}\Big)\, R_D \;, \tag{2.30}$$

$$\begin{aligned}\mathbb{E}(R_M) = {} & p^f_{R_M}\Big(\mathbb{E}(T^{\text{lost}}_{R_M}) + \mathbb{E}(R_D) + \mathbb{E}(R_M) + \mathbb{E}(T^{\text{rec}})\Big) \\ & + \Big(1 - p^f_{R_M}\Big)\, R_M \;,\end{aligned} \tag{2.31}$$

$$\begin{aligned}\mathbb{E}(C_D) = {} & p^f_{C_D}\Big(\mathbb{E}(T^{\text{lost}}_{C_D}) + \mathbb{E}(R_D) + \mathbb{E}(R_M) + \mathbb{E}(T^{\text{rec}}) + \mathbb{E}(C_M) + \mathbb{E}(C_D)\Big) \\ & + \Big(1 - p^f_{C_D}\Big)\, C_D \;,\end{aligned} \tag{2.32}$$

$$\begin{aligned}\mathbb{E}(C_M) = {} & p^f_{C_M}\Big(\mathbb{E}(T^{\text{lost}}_{C_M}) + \mathbb{E}(R_D) + \mathbb{E}(R_M) + \mathbb{E}(T^{\text{rec}}) + \mathbb{E}(C_M)\Big) \\ & + \Big(1 - p^f_{C_M}\Big)\, C_M \;,\end{aligned} \tag{2.33}$$

where $\mathbb{E}(T^{\text{rec}})$ denotes the expected time to re-execute the whole pattern, or part of it, depending on when the fault strikes. If the fault strikes during disk checkpointing (Equation (2.32)), the entire pattern needs to be re-executed. But if the fault strikes during memory checkpointing (Equation (2.33)), then only part of the pattern up to the given memory checkpoint needs to be re-executed. In all cases, $\mathbb{E}(T^{\text{rec}})$ is upper bounded by the expected execution time of the whole pattern. Recall from our previous analysis that the optimal pattern length satisfies $W^* = \Theta\left(\lambda^{-1/2}\right)$ and the optimal overhead satisfies $H^*(\text{P}) = \Theta\left(\lambda^{1/2}\right)$. Hence, in an optimized pattern, we will have $\mathbb{E}(T^{\text{rec}}) \leq \mathbb{E}(\text{P}) = W^* + H^*(\text{P}) \times W^* = \Theta\left(\lambda^{-1/2}\right)$. Solving Equations (2.30) to (2.33), we can then derive the following results:

$$\begin{aligned}\mathbb{E}(R_D) &= R_D + O(\lambda) \;, \\ \mathbb{E}(R_M) &= R_M + O(\sqrt{\lambda}) \;, \\ \mathbb{E}(C_D) &= C_D + O(\sqrt{\lambda}) \;, \\ \mathbb{E}(C_M) &= C_M + O(\sqrt{\lambda}) \;,\end{aligned}$$

which suggest that the expected costs to perform checkpoints and recoveries are dominated by their original costs under the assumption of a large platform MTBF. Intuitively, this is due to the small probability of encountering an error during these operations. Thus, in Propositions 2 to 5, replacing C_D, C_M, R_D and R_M by their expected values does not affect the expected execution times of the patterns in the first-order approximation.

Now, we briefly discuss the impact of having errors during verifications. Basically, as far as fail-stop errors are concerned, we can consider any (partial or guaranteed) verification together with the work segment (or chunk) immediately preceding it. Two expressions need to be modified in the analysis. First, the probability of experiencing at least one error during the execution of any segment (or chunk) of length w becomes $p^f = 1 - e^{-\lambda_f(w+V)}$, where V denotes the cost of the verification at the end of the work length w. Second, the expected time loss

in executing this segment (or chunk) becomes $\mathbb{E}(T^{\text{lost}}) = \frac{1}{\lambda_f} - \frac{w+V}{e^{\lambda_f(w+V)}-1}$. It turns out that both changes do not affect the first-order approximation, because the extra terms (involving V) contribute $O(\sqrt{\lambda})$ to the expected execution time of a pattern, which again is negligible compared to the dominant terms under a large platform MTBF.

2.6 Performance evaluation

In this section, we conduct a set of simulations whose goal is twofold: (i) corroborate the theoretical study, and (ii) assess the relative efficiency of each checkpoint and verification type under realistic scenarios. We rely on simulations to evaluate the performance of the computing patterns at extreme scale, and we instantiate the model with three scenarios. In the first scenario, we evaluate the performance of each pattern using real parameters from the literature. The second scenario is a weak scaling experiment, whose purpose is to assess the scalability of the approach on increasingly large platforms. In the last scenario, we study the impact of varying error rates on the overhead of the method. The simulator code is publicly available at `http://graal.ens-lyon.fr/~yrobert/two-level`, so interested readers can experiment with it and build relevant scenarios of their choice.

2.6.1 Simulation setup

We make several assumptions on the input parameters. First, we assume that the recovery cost is equivalent to the corresponding checkpointing cost, i.e., $R_D = C_D$ and $R_M = C_M$. This is reasonable because writing a checkpoint and reading one typically takes the same amount of time. Then, we assume that a guaranteed verification must check all the data in memory, making its cost in the same order as that of a memory checkpoint, i.e., $V^* = C_M$. Furthermore, we assume partial verifications similar to those proposed in [8, 10, 15], with very low cost while offering good recalls. In the following, we set $V = \frac{V^*}{100}$ and $r = 0.8$. All these choices are somewhat arbitrary and can easily be modified in the simulator; we believe they represent reasonable values for current and next-generation HPC applications.

The simulator generates errors following an exponential distribution of parameter λ_f for fail-stop errors and λ_s for silent errors. The simulator allows fail-stop errors to occur during computations, verifications, checkpoints and recoveries, while silent errors are only allowed during actual computations, which is in accordance with our model.

An experiment goes as follows. We feed the simulator with the description of the platform, consisting of the parameters λ_f, λ_s, C_D and C_M (since the other parameters can be deduced from the above assumptions). For each pattern, we compute the optimal length W^*, the optimal overhead $H^*(\text{P})$, as well as the optimal number of memory checkpoints n^* and the optimal number of verifications $\boldsymbol{m}^*$ (when applicable), using the formulas from Table I. The total amount of work for the application is set to that of 1000 optimal patterns, and the simulator runs each experiment 1000 times. For each pattern, it outputs the simulated overhead, the simulated number of disk checkpoints, memory checkpoints, verifications, disk recoveries and memory recoveries by averaging the values from the 1000 runs.

Table II
PLATFORM PARAMETERS

platform	#nodes	λ_f	λ_s	C_D	C_M
Hera	256	9.46e-7	3.38e-6	$300s$	$15.4s$
Atlas	512	5.19e-7	7.78e-6	$439s$	$9.1s$
Coastal	1024	4.02e-7	2.01e-6	$1051s$	$4.5s$
Coastal SSD	1024	4.02e-7	2.01e-6	$2500s$	$180.0s$

2.6.2 Assessing resilience mechanisms on real platforms

In the first scenario, we assess the performance of the optimal patterns on four different platforms with real parameter settings. We compare the results for the six optimal patterns of Table I.

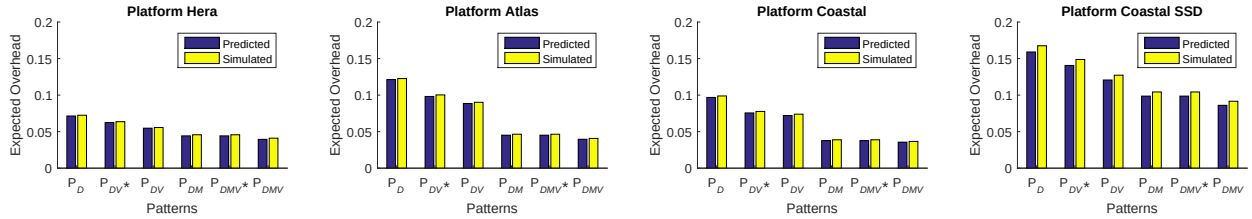
Platform settings

Table II presents the four platforms used in this experiment and their main parameters. These platforms have been used to evaluate the Scalable Checkpoint/Restart (SCR) library by Moody et al. [74], who provide accurate measurements for λ_f, λ_s, C_D and C_M using real applications. Note that the Hera platform has the worst error rates, with a platform MTBF of 12.2 days for fail-stop errors and 3.4 days for silent errors. In comparison, and despite its higher number of nodes, the Coastal platform features a platform MTBF of 28.8 days for fail-stop errors and 5.8 days for silent errors. In addition, the last platform uses SSD technology for memory checkpointing, which provides more data space, at the cost of higher checkpointing costs.

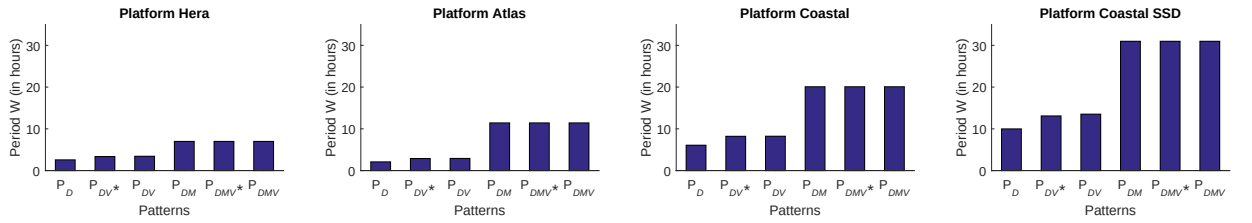
Pattern overhead

Figure 2.6a presents, for each pattern, the predicted overhead $H^*(\mathrm{P})$ (in blue) versus the simulated one (in yellow) on each platform. Remember that the formula used to compute the expected overhead is the result of a first-order approximation, meaning that we are ignoring some low-order terms in the computation. The consequence is that the predicted overhead, being a little optimistic, is always slightly inferior compared to the simulated one. However, the difference between both overheads is very small (less than 1%), which validates the model quite satisfactorily.

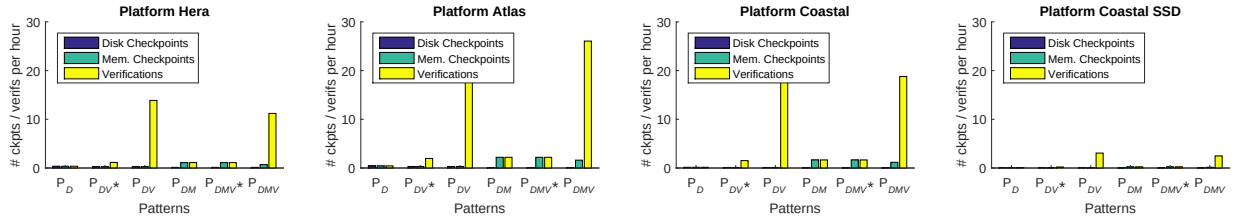
Overall, the overhead oscillates between 4% and 7% on Hera, where checkpoints are relatively cheaper, to just over 15% on Coastal SSD, where checkpoints are more expensive. Regardless of the platform, the more advanced patterns always result in smaller overheads. In particular, we observe a significant difference between the first three patterns (P_D, P_{DV^*}, P_{DV}), which use single-level checkpointing, and the last three patterns (P_{DM}, P_{DMV^*}, P_{DMV}), which use two-level checkpointing. The gap is more visible for Atlas (5%) and Coastal (4%), where the difference between the costs of a disk checkpoint and a memory checkpoint is larger, thus making memory checkpoints more valuable.



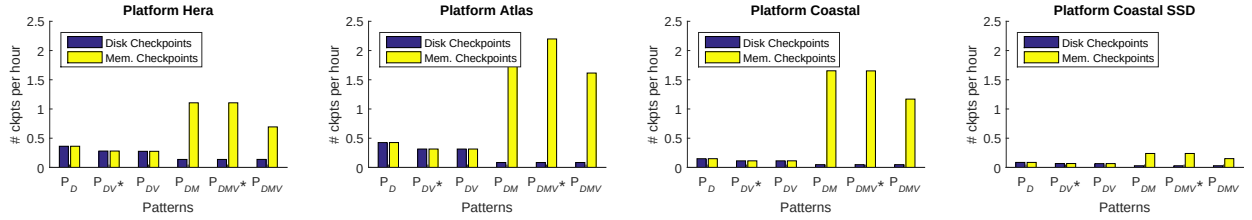
(a)



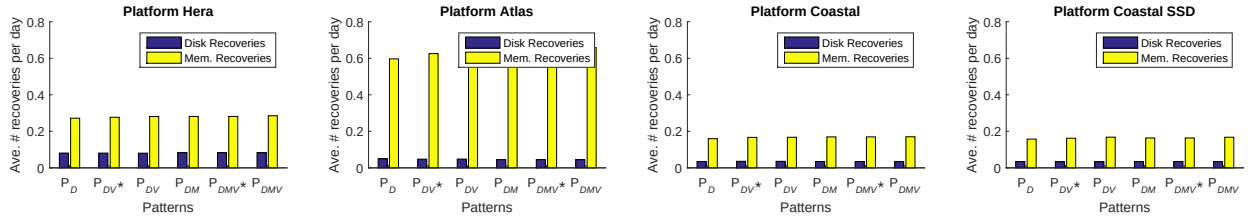
(b)



(c)



(d)



(e)

Figure 2.6: Performance of different patterns on the four platforms. Each column represents one platform.

Pattern periods

Figure 2.6b shows the work lengths (periods) of the patterns on each platform. We observe that single-level patterns are associated with shorter periods (around 3 hours on Hera and 10 hours on Coastal), as opposed to the longer periods shown by the two-level patterns (around 8 hours on Hera and 20 hours on Coastal). Indeed, when a fail-stop error strikes, the only choice is to recover from the last disk checkpoint, losing all the work done so far. In that case, a short period helps to mitigate the amount of time lost.

Nevertheless, silent errors are more prominent on these platforms and when one occurs, two-level patterns can recover from an intermediate memory checkpoint instead. Not only does that provide a faster recovery, but also it does not require the application to restart from the very beginning of the pattern. As a result, disk checkpoints are only used for fail-stop errors, and a longer period is favored in order to accommodate more but cheaper intermediate memory checkpoints.

Pattern checkpoints

Figure 2.6c presents the average number of disk checkpoints, memory checkpoints and verifications taken each day by each pattern. We take all checkpoints and verifications from the simulations into account, including the ones performed in recoveries and re-executions. Since a partial verification is much cheaper than a guaranteed one, the two patterns that are allowed to use them (P_{DV} and P_{DMV}) tend to take full advantage of this mechanism. On Hera, P_{DV} generates an average of 13 verifications per hour (including the guaranteed ones), which is slightly more than its two-level counterpart (P_{DMV}), which generates 12 verifications per hour. On Coastal, there are more than 20 verifications per hour for P_{DV} and 19 for P_{DMV}. Note that the verifications before each memory checkpoint are included.

In order to have a closer look at the number of checkpoints, which is dwarfed by the number of verifications, Figure 2.6d presents the checkpointing frequencies alone. Naturally, for the two-level patterns, whose periods are longer, the disk checkpointing frequencies are smaller. However, their memory checkpointing frequencies are higher, because the cheaper memory checkpoints are favored in these two-level schemes, in order to better protect the application from silent errors. Note that the memory checkpoints before each disk checkpoint are also taken into account. Lastly, we observe that the Coastal SSD platform requires very few verifications and memory checkpoints. This is because the cost of a memory checkpoint is much higher on this platform ($180s$) as opposed to the costs on other platforms ($15.4s$ on Hera and $4.5s$ on Coastal).

Pattern recoveries

Finally, Figure 2.6e shows the number of recoveries per day for each pattern on each platform. We can see that the number of disk recoveries follows closely the fail-stop error rate of a given platform, and it is not affected by the patterns. Indeed, when a fail-stop error strikes, a disk recovery is performed regardless of the pattern used. On Hera, we observe 0.083 disk recovery per day on average, translating to approximately one recovery every 12 days, which is in accordance with the platform MTBF of 12.2 days for fail-stop errors. The same applies to

Atlas and Coastal, which show respectively 0.044 and 0.034 disk recoveries per day on average (equivalent to a platform MTBF of 22 days for Atlas and 29 days for Coastal).

The number of memory recoveries may be influenced by several factors. This is because a memory recovery is not performed immediately after the occurrence of a silent error. Instead, it is performed only when an alarm is raised by a verification, or when a fail-stop error strikes. In both cases, more than one silent error could have occurred before the memory recovery. In the latter case, a memory recovery is triggered right after a disk recovery, possibly without any silent error. In general, the memory recovery frequency could well depend on whether partial verifications are used in a pattern and the length of the pattern. This also explains the slight difference under different patterns. Nevertheless, the simulation results show that the silent error rate is a good indicator of the memory recovery frequency. For instance, on Hera, we observe 0.285 memory recovery per day on average, which is approximately one memory recovery every 3.5 days. This is very close to the MTBF of 3.4 days for silent errors.

2.6.3 Weak scaling experiment

In order to assess the scalability of the model, we now present the results of a weak scaling experiment. This experiment is based on the Hera platform, whose disk checkpoint cost is the closest to state-of-the-art platforms (5 minutes).

Platform settings

We first calculate the MTBF of one computing node, namely 8.57 years for fail-stop errors and 2.4 years for silent errors. The platform MTBF is obtained by dividing the per-node MTBF by the number of nodes used in the simulation. For example, when 2^{17} nodes are used, the MTBF decreases to about $2064s$ for fail-stop errors and $577s$ for silent errors.

Under weak scaling, the problem size grows linearly with the number of nodes, so the time needed to perform a memory checkpoint C_M remains constant. In addition, we make the optimistic assumption that the cost of a disk checkpoint C_D also remains constant by scaling the I/O bandwidth of the file system[3]. We explore two scenarios. In the first scenario, we set the cost of a disk checkpoint to be the same as on Hera, i.e., $300s$. In the second scenario, we reduce the cost of a disk checkpoint to $90s$ to account for improved disk technology.

Results

Figure 2.7a presents the impact of the number of nodes on the overheads for the simplest pattern P_D and the most advanced pattern P_{DMV}. From the simulation results, we can see that the performance remains acceptable up to $2^{15} = 32768$ nodes, with an overhead of 100% for P_D and 64% for P_{DMV}. Beyond that number, the overhead increases drastically for both patterns, eventually exceeding 500% for $2^{18} = 262144$ nodes. However, compared to the simple pattern P_D, the two-level pattern P_{DMV} improves the overhead by a few percent on 256 nodes up to over 150% on 2^{18} nodes.

[3]In actual systems, the I/O bandwidth could become a bottleneck, resulting in increased disk checkpointing cost. This would further widen the performance gap between single-level and two-level patterns.

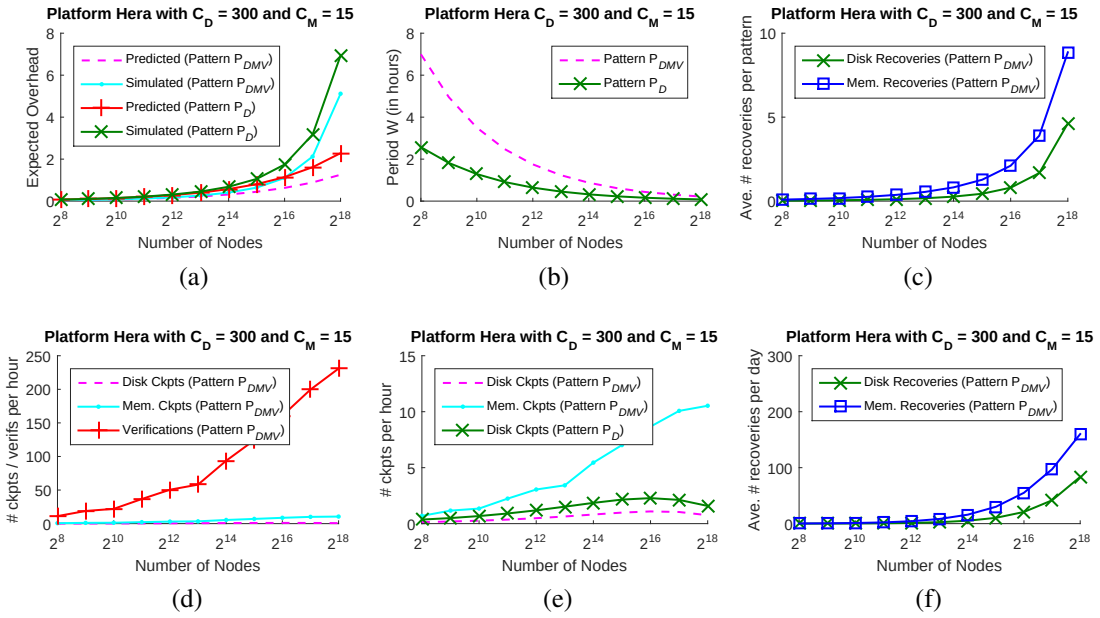


Figure 2.7: Weak scaling experiment on the Hera platform.

We also observe the difference between the simulated overhead and the predicted one, which starts negligible for a small number of nodes but reaches more than a factor of three for 2^{18} nodes. The reason is the use of first-order approximation to compute the predicted overhead, which is only accurate when the platform MTBF is large in front of the other parameters. Obviously, this is no longer the case for a large number of nodes. For instance, when the number of nodes reaches 10^5 (almost 2^{17} nodes), the MTBF of the whole platform reduces down to less than 10 minutes, which is in the same order as the period of a pattern (Figure 2.7b). At this point, the application experiences a few errors per pattern (Figure 2.7c) and nearly a dozen errors per hour (Figure 2.7f). In order to minimize the impact of the errors, the pattern P_{DMV} places more than 200 verifications and more than 10 memory checkpoints per hour (Figures 2.7e and 2.7d). As a result, a lot of time is wasted on resilience operations, and the model starts to show its limits. However, when the error rate is this high, there is not much flexibility left in the optimization, and no pattern is able to offer satisfactory performance.

Finally, Figure 2.8 presents the results when we repeat the weak scaling experiment with a disk checkpointing cost of $90s$ instead of $300s$. Since writing a disk checkpoint is cheaper now, the period is reduced and checkpointing frequency is increased. Overall, the overheads become much better, around 200% at 2^{18} nodes, as opposed to 500% observed in Figure 2.7a. The behavior of other parameters is similar to the ones presented in Figure 2.7.

2.6.4 Impact of error rates

We assess the impact of the error rates on the performance of the computing patterns. Again, we focus on the Hera platform but scale its number of nodes to 10^5. We vary the error rates λ_f and λ_s with respect to their nominal values while keeping the other parameters fixed.

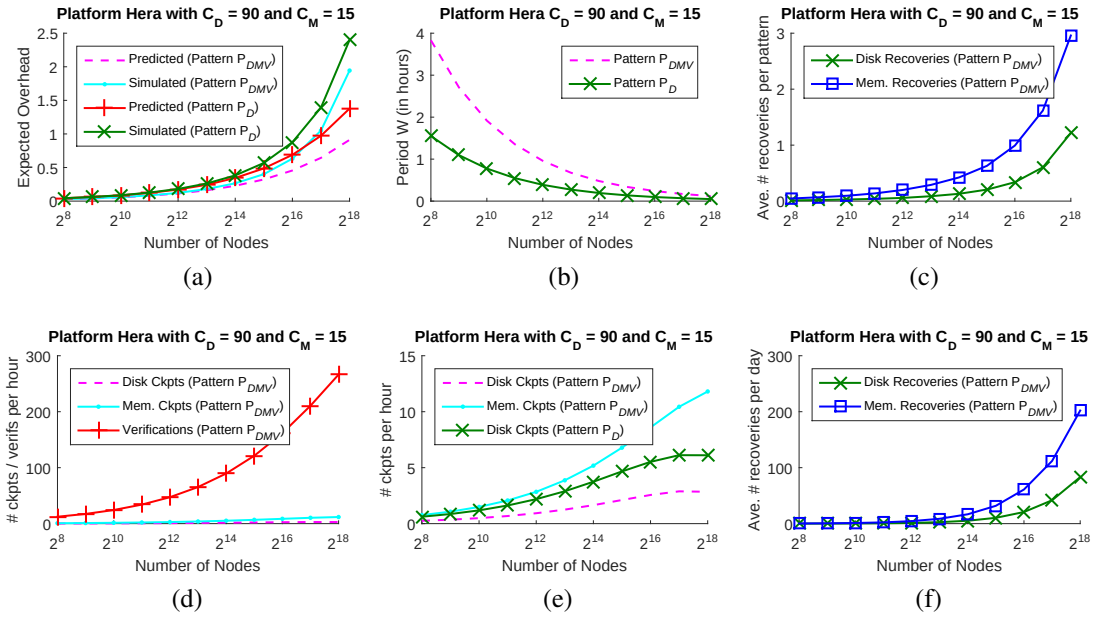


Figure 2.8: Weak scaling experiment on the Hera platform with reduced disk checkpointing cost.

Figures 2.9a and 2.9b present the impact of λ_f and λ_s on the simulated overheads of the two patterns P_D and P_{DMV}. For the P_{DMV} pattern, we observe that the overhead is affected more by the fail-stop errors than by the silent errors. This is because the intermediate memory checkpoints better protect the application from silent errors. On the other hand, the overhead of the single-level pattern P_D is affected more by the silent errors, simply because silent errors have a much higher rate. Figure 2.9c shows the difference between the overheads of both patterns. We observe a similar performance when most errors are fail-stop, due to their relatively small rate. However, when the silent error rate increases, the two-level pattern achieves a much better performance than the single-level pattern, by saving up to 200% on the execution overhead.

We now study the impact of error rates on the checkpointing period and frequency. Figure 2.9d presents the impact of fail-stop errors on the period of both patterns when the silent error rate is fixed at its nominal value. We can see that the period for P_D remains constant, while the period for P_{DMV} decreases with increased λ_f. This is because the high silent error rate has already driven the P_D pattern period very low (< 10 minutes), so increasing the fail-stop error rate has a limited impact. On the other hand, the period for the P_{DMV} pattern is primarily driven by the fail-stop error rate, so it decreases quickly, allowing more disk checkpoints to be taken. In addition, Figures 2.9e and 2.9f show that the number of checkpoints successfully taken in each hour remain stable for both patterns. Since the period of P_{DMV} decreases while the period of P_D remains constant, it implies degraded performance for the two-level pattern and stable performance for the single-level one, corroborating the previous analysis. Figure 2.9g shows the corresponding number of recoveries, which is again in accordance with the MTBF of the platform. Note that the number of memory recoveries decreases slightly, as some silent errors

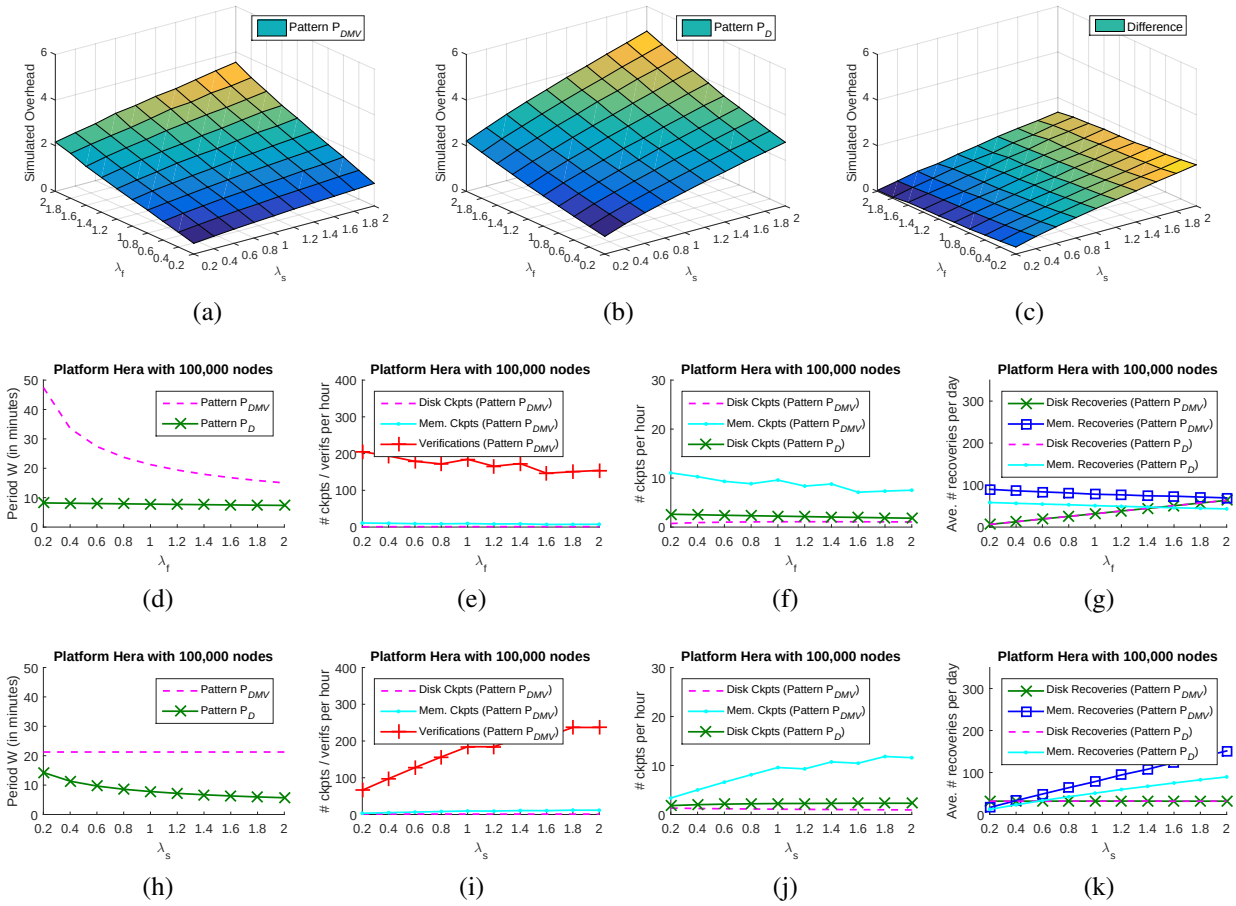


Figure 2.9: Impact of error rates λ_f and λ_s on the performance on the Hera platform with 10^5 nodes.

are masked by fail-stop errors.

Figure 2.9h shows the impact of silent errors on the performance of both patterns when the fail-stop error rate is fixed at the nominal value. Now, the role is reversed. Since the P_{DMV} pattern is equipped with more memory checkpoints and verifications, silent errors have no impact on the period. On the contrary, the period of the P_D pattern decreases in order to detect silent errors earlier, which is the only way to protect the application from increased silent error rate. As shown by Figures 2.9i and 2.9j, the number of verifications and memory checkpoints performed by P_{DMV} increases with the silent error rate in order to compensate for the fixed number of disk checkpoints. For the P_D pattern, the checkpointing frequency remains the same, implying degraded performance with decreased period. Finally, Figure 2.9k shows the corresponding number of recoveries. When silent errors are more prominent, two-level checkpointing helps to detect them before the end of the pattern, as shown by the higher number of memory recoveries. This results in faster recoveries and overall better performance.

2.6.5 Summary

Through the simulation results of this section, we conclude that the first-order approximation for the resilience patterns provides an accurate performance model for systems with up to tens of thousands of nodes. Overall, the complex pattern that combines all resilience mechanisms offers significantly better performance, improving the base pattern by up to 150% in the execution overhead. The findings are consistent on different platforms and with varying error rates. The results nicely corroborate the analytical study, and demonstrate the benefit of using two-level patterns for dealing with both fail-stop and silent errors.

2.7 Related work

2.7.1 Checkpointing

The most commonly deployed strategy to cope with fail-stop errors is checkpointing, in which processes periodically save their state, so that computation can be resumed from that point when some failure disrupts the execution. Checkpointing strategies are numerous, ranging from fully coordinated checkpointing [28] to uncoordinated checkpoint and recovery with message logging [46]. Despite a very broad applicability, all these fault-tolerance methods suffer from the intrinsic limitation that both protection and recovery generate an I/O workload that grows with failure probability, and becomes unsustainable at large scale [17, 52] (even when considering optimizations such as diskless or incremental checkpointing [78]).

To reduce the checkpointing overhead, many authors have proposed multi-level checkpointing protocols, which combine global disk checkpointing with local or in-memory checkpointing (also known as *diskless checkpointing*). Most of these protocols handle fail-stop errors only, and recover from different levels of checkpoints according to the severity of the failures. Vaidya [94] proposed a two-level recovery scheme that tolerates a single node failure using a local checkpoint stored on a parter node. If more than one failure occurs during any local checkpointing interval, the scheme then resorts to the global checkpoint. Silva and Silva [88]

advocated a similar scheme by using memory to store the local checkpoints, which is protected by XOR encoding. Moody et al. [74] generalized this idea to account for an arbitrary number of levels with increasing failure handling capability, and implemented it in a Scalable Checkpoint/Restart (SCR) library. Bautista-Gomez et al. [12] also designed a multi-level checkpointing library, called Fault Tolerance Interface (FTI), but they employed a more efficient Reed-Solomon encoding scheme to handle multiple failures without the need to access the parallel file system.

Our work is along the same direction as multi-level checkpointing, but the two levels we propose target different error sources, namely, fail-stop errors and silent errors. As mentioned before, this dramatically changes the computation of the expected re-execution time, because we do not have to distinguish which error type strikes first. Moreover, as in Young [97] and Daly [36], we provide explicit formulas on the optimal checkpointing intervals for both levels (up to a first-order approximation), while previous work relies on numerical methods to find the optimal solution [37].

2.7.2 Silent error detection

Considerable efforts have been directed at verification techniques to reveal silent errors. A guaranteed verification is often only achievable with expensive techniques, such as process replication [53, 75] or redundancy [45, 72]. Application-specific information can be very useful in decreasing the verification cost. Algorithm-based fault tolerance (ABFT) [16, 63, 87] is a well-known technique to detect errors in linear algebra kernels using checksums. Various techniques have been proposed in other application domains. Benson et al. [14] compared a higher-order scheme with a lower-order one to detect errors in the numerical analysis of ODEs. Sao and Vuduc [85] investigated self-stabilizing corrections after error detection in the conjugate gradient method. Heroux and Hoemmen [19] designed a fault-tolerant GMRES capable of converging despite silent errors. Bronevetsky and de Supinski [21] provided a comparative study of detection costs for iterative methods. Recently, detectors based on data analytics have been proposed to serve as partial verifications [8, 10, 15]. These detectors use interpolation techniques, such as time series prediction and spatial multivariate interpolation, on scientific dataset to offer large error coverage for a negligible overhead. Although not perfect, their accuracy-to-cost ratios tend to be very high, which makes them interesting alternatives at large scale.

2.7.3 Optimization of computing patterns

Given the checkpointing cost and the platform MTBF, classical formulas due to Young [97] and Daly [36] are well known to determine the optimal checkpointing intervals in the presence of fail-stop errors. These formulas have been extended to account for silent errors in various ways. By coupling checkpointing with guaranteed verification, Aupy et al. [3] analyzed two simple patterns: one with k checkpoints and one verification, and the other with k verifications and one checkpoint. In a previous work, Benoit et al. [J3] studied the latter pattern and gave explicit formulas to accommodate both fail-stop and silent errors. The idea of interleaving p checkpoints and q verifications has also been explored in [13] to achieve more

optimized computing patterns. A first analysis of a pattern that utilizes partial verification for silent error detection was given in Chapter 1. To the best of our knowledge, this work is the first to investigate the combination of in-memory checkpoints, disk checkpoints, partial verifications and guaranteed verifications.

2.8 Conclusion

When computing at extreme scale, both fail-stop errors and silent errors are major threats to executing HPC applications with acceptable overhead. While several techniques have been developed to cope with either threat, few approaches are devoted to addressing both of them simultaneously. Although surprising –because dealing with both error sources is unavoidable on large-scale platforms–, this lack of solutions may be explained by the new challenges raised by silent errors, whose detection is not immediate and requires the use of verification mechanisms, either partial or guaranteed. Also, the interplay of two levels of checkpoints and of two types of verifications raises difficult optimization challenges. The major contribution of this chapter is the characterization of the optimal computational pattern. The derivation is technically involved, but the results are easy to use in real-life scenarios: one has just to look at Table I and pick the optimal pattern that fits their resilience needs.

The accuracy of our model as well as the analysis have been nicely corroborated by extensive simulations. The results show acceptable difference in the predicted overhead and the simulated one on systems with up to tens of thousands of nodes. Also, the complex pattern that combines all resilience mechanisms provides up to 150% improvement in the execution overhead compared to the base pattern dictated by the classical Young/Daly's formula.

Chapter 3

Towards Optimal Multi-Level Checkpointing with Fail-Stop Errors

As opposed to previous chapters, this work focuses on fail-stop errors only. The problem is similar to that of Chatpter 2, however dealing with more than two levels of checkpoint makes the analysis much more challenging. We provide a framework to analyze multi-level checkpointing protocols, by formally defining a k-level checkpointing pattern. We provide a first-order approximation to the optimal checkpointing period, and show that the corresponding overhead is in the order of $\sum_{\ell=1}^{k} \sqrt{2\lambda_\ell C_\ell}$, where λ_ℓ is the error rate at level ℓ, and C_ℓ the checkpointing cost at level ℓ. This nicely extends the classical Young/Daly formula on single-level checkpointing. Furthermore, we are able to fully characterize the shape of the optimal pattern (number and positions of checkpoints), and we provide a dynamic programming algorithm to determine the optimal subset of levels to be used. Finally, we perform simulations to check the accuracy of the theoretical study and to confirm the optimality of the subset of levels returned by the dynamic programming algorithm. The results nicely corroborate the theoretical study, and demonstrate the usefulness of multi-level checkpointing with the optimal subset of levels. The work in this chapter has been published in *Transactions on computers* (TC) [J2].

3.1 Introduction

Checkpointing is the de-facto standard resilience method for HPC platforms at extreme-scale. However, the traditional single-level checkpointing method suffers from significant overhead, and multi-level checkpointing protocols now represent the state-of-the-art technique. These protocols allow different levels of checkpoints to be set, each with a different checkpointing overhead and recovery ability. Typically, each level corresponds to a specific fault type, and is associated to a storage device that is resilient to that type. For instance, Chapter 2 deals with a two-level approach: one type of checkpoint to handle transient memory errors (level 1) by storing key data in main memory; and another to address node failures (level 2) by storing key data in stable storage (remote redundant disks).

In this chapter, we deal with fail-stop errors only. We consider a very general scenario, where the platform is subject to k levels of faults, numbered from 1 to k. Level ℓ is associated

with an error rate λ_ℓ, a checkpointing cost C_ℓ, and a recovery cost R_ℓ. A fault at level ℓ destroys all the checkpoints of lower levels (from 1 to $\ell - 1$ included) and implies a roll-back to a checkpoint of level ℓ or higher. Similarly, a recovery of level ℓ will restore data from all lower levels. Typically, fault rates are decreasing and checkpoint/recovery costs are increasing when we go to higher levels: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k$, $C_1 \leq C_2 \leq \cdots \leq C_k$, and $R_1 \leq R_2 \leq \cdots \leq R_k$.

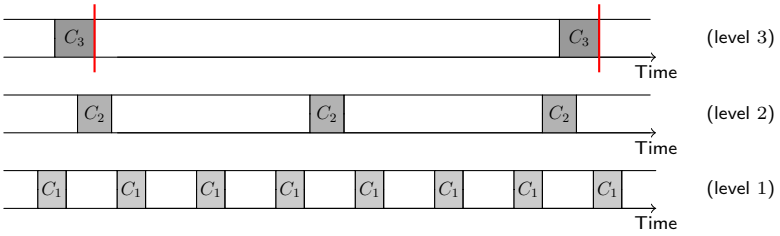


Figure 3.1: Independent checkpointing periods for three levels of faults: no synchronization between checkpoint levels.

The idea of multi-level checkpointing is that checkpoints are taken for each level of faults, but at different periods. Intuitively, the less frequent the faults, the longer the checkpointing period: this is because the risk of a failure striking is lower when going to higher levels; hence the expected re-execution time is lower too; one can safely checkpoint less frequently, thereby reducing failure-free overhead (checkpointing is useless in the absence of fault). There are several natural approaches to implement multi-level checkpointing. The first option is to use *independent checkpointing periods* for each level, as illustrated in Figure 3.1 with $k = 3$ levels. This option raises several difficulties, the most prominent one being overlapping checkpoints. Typically, we need to checkpoint different levels in sequence (e.g., writing into memory before writing onto disk), so we would need to delay some checkpoints, which might not be possible in some environments, and which would introduce irregular periods. The second option is to synchronize all checkpoint levels by nesting them inside a *periodic pattern* that repeats over time, as illustrated in Figure 3.2(a). In this figure, the **pattern** has five computational **segments**, each followed by a level-1 checkpoint. A segment is a chunk of work between two checkpoints, and a pattern consists in segments and checkpoints. The second and fifth level-1 checkpoints are followed by a level-2 checkpoint. Finally, the pattern ends with a level-3 checkpoint. When using patterns, a checkpoint at level ℓ is always preceded by checkpoints at all lower levels 1 to $\ell - 1$, which makes good sense in practice (e.g., with two levels, main memory and disk, one writes the data into memory before transferring it to disk).

Using periodic patterns simplifies the orchestration of checkpoints at all levels. In addition, repeatedly applying the same pattern is optimal for on-line scheduling problems, or for jobs running a very long (even infinite) time on the platform. Indeed, in this scenario, we seek the best pattern, i.e., the one whose overhead is minimal. The *overhead* of a pattern is the price per work unit to pay for resilience in the pattern; hence minimizing overhead is equivalent to optimizing platform throughput. For a pattern $\mathrm{P}(W)$ with W units of work (the cumulated length of all its segments), the overhead $H(\mathrm{P(W)})$ is defined as the ratio of the pattern's expected ex-

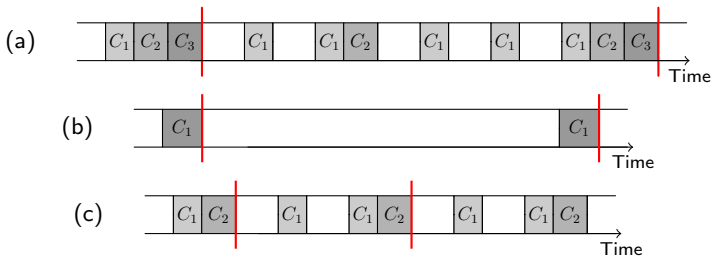


Figure 3.2: Checkpointing patterns (highlighted using red bars) with (a) $k = 3$, (b) $k = 1$, and (c) $k = 2$ levels.

ecution time $\mathbb{E}(\mathrm{P(W)})$ over its total work W minus 1:

$$H(\mathrm{P}(W)) = \frac{\mathbb{E}(\mathrm{P}(W))}{W} - 1. \tag{3.1}$$

If there were neither checkpoint nor fault, the overhead would be zero. Determining the optimal pattern (with minimal overhead), and then repeatedly using it until job completion, is the optimal approach with Exponential failure distributions and long-lasting jobs. Indeed, once a pattern is successfully executed, the optimal strategy is to re-execute the same pattern. This is because of the memoryless property of exponential distributions: the history of failures has no impact on the solution, so if a pattern is optimal at some point in time, it stays optimal later in the execution, because we have no further information about the amount of work still to be executed.

The difficulty of characterizing the optimal pattern dramatically increases with the number of levels. How many checkpoints of each level should be used, and at which locations inside the pattern? What is the optimal length of each segment? With one single level (see Figure 3.2(b)), there is a single segment of length W, and the Young/Daly formula [36, 97] gives $W^{\mathrm{opt}} = \sqrt{\frac{2C_1}{\lambda_1}}$. The minimal overhead is then $H^{\mathrm{opt}} = \sqrt{2\lambda_1 C_1} + O(\lambda_1)$.

With two levels, the pattern still has a simple shape, with N segments followed by a level-1 checkpoints, and ended by a level-2 checkpoint (see Figure 3.2(c)). Recent work [39] shows that all segments have same length in the optimal pattern, and provides mathematical equations that can be solved numerically to compute both the optimal length W^{opt} of the pattern and its optimal number of segments. However, no closed-form expression is available, neither for W^{opt}, nor for the minimal overhead H^{opt}.

With three levels, no optimal solution is known. The pattern shape becomes quite complicated. Coming back to Figure 3.2(a), we identify two sub-patterns ending with a level-2 checkpoint. The first sub-pattern has 2 segments while the second one has 3. The memoryless property does not imply that all sub-patterns are identical, because the state after completing the first sub-pattern is not the same as the initial state when beginning the execution of the pattern. In the general case with k levels, the shape of the pattern will be even more complicated, with different-shaped sub-patterns (each ended by a level $k-1$ checkpoint). In turn, each sub-pattern may have different-shaped sub-sub-patterns (each ended by a level $k-2$ checkpoint), and so on. The major contribution of this work is to provide an analytical characterization of the

optimal pattern with an arbitrary number k of checkpointing levels, with closed-form formulas for the pattern length W^{opt}, the number of checkpoints at each level, and the optimal overhead H^{opt}. In particular, we obtain the following beautiful result:

$$H^{\text{opt}} = \sum_{\ell=1}^{k} \sqrt{2\lambda_\ell C_\ell} + O(\Lambda), \tag{3.2}$$

where $\Lambda = \sum_{\ell=1}^{k} \lambda_\ell$. However, we point out that this analytical characterization relies on a first-order approximation, so it is valid only when resilience parameters C_ℓ and R_ℓ are small in front of the platform Mean Time Between Failures (MTBF) $\mu = 1/\Lambda$. Also, the optimal pattern has rational number of segments, and we use rounding to derive a practical solution. Still, Equation (3.2) provides a lower bound on the optimal overhead, and this bound is met very closely in all our experimental scenarios.

Finally, in many practical cases, there is no obligation to use all available checkpointing levels. For instance, with $k = 3$ levels, one may choose among four possibilities: level 3 only, levels 1 and 3, levels 2 and 3, and all levels 1, 2 and 3. Of course, we still have to account for all failure types, which translates into the following:

- level 3: use $\lambda_3 \leftarrow \lambda_1 + \lambda_2 + \lambda_3$;
- levels 1 and 3: use λ_1 and $\lambda_3 \leftarrow \lambda_2 + \lambda_3$;
- levels 2 and 3: use $\lambda_2 \leftarrow \lambda_1 + \lambda_2$ and λ_3;
- all levels: use λ_1, λ_2 and λ_3.

Our analytical characterization of the optimal pattern leads to a simple dynamic programming algorithm for selecting the optimal subset of levels.

The rest of this chapter is organized as follows. Section 3.2 surveys the related work. Section 3.3 is the heart of the chapter and shows how to compute the optimal pattern as well as the optimal subset of levels. Section 3.4 is devoted to simulations assessing the accuracy of the first-order approximation. Finally, Section 3.5 provides concluding remarks and hints for future work.

3.2 Related work

Given the checkpointing cost and platform MTBF, classical formulas due to Young [97] and Daly [36] are well known to determine the optimal checkpointing period in the single-level checkpointing scheme. However, this method suffers from the intrinsic limitation that the cost of checkpointing/recovery grows with failure probability, and becomes unsustainable at large scale [17, 52] (even with diskless or incremental checkpointing [78]).

To reduce the I/O overhead, various two-level checkpointing protocols have been studied. Vaidya [94] proposed a two-level recovery scheme that tolerates a single node failure using a local checkpoint stored on a parter node. If more than one failure occurs during any local

checkpointing interval, the scheme resorts to the global checkpoint. Silva and Silva [88] advocated a similar scheme by using memory to store local checkpoints, which is protected by XOR encoding. Di et al. [39] analyzed a two-level checkpointing pattern, and proved equal-length segments in the optimal solution. They also provided mathematical equations that can be solved numerically to compute the optimal pattern length and number of segments. In Chapter 2, we relied on disk checkpoints to cope with fail-stop failures and memory checkpoints coupled with error detectors to handle silent data corruptions. We have derived first-order approximation formulas for the optimal pattern length and the number of memory checkpoints between two disk checkpoints.

Some authors have also generalized two-level checkpointing to account for an arbitrary number of levels. Moody et al. [74] implemented this approach in a three-level Scalable Checkpoint/Restart (SCR) library. They relied on a rather complex Markov model to recursively compute the efficiency of the scheme. Bautista-Gomez et al. [12] designed a four-level checkpointing library, called Fault Tolerance Interface (FTI), in which partner-copy and Reed-Solomon coding are employed as two intermediate levels between local and global disks. Based on FTI, Di et al. [37] proposed an iterative method to compute the optimal checkpointing interval for each level with prior knowledge of the application's total execution time. Hakkarinen and Chen [60] considered multi-level diskless checkpointing for tolerating simultaneous failures of multiple processors. Balaprakash et al. [6] studied the trade-off between performance and energy for general multi-level checkpointing schemes.

While all of these works relied on numerical methods to compute the checkpointing intervals at different levels, this work is the first to provide explicit formulas on the optimal parameters in a multi-level checkpointing protocol (up to first-order approximation as in Young/Daly's classical result).

3.3 Computing the optimal pattern

This section computes the optimal multi-level checkpointing pattern. We first state our assumptions in Section 3.3.1, and then analyze the simple case with $k = 2$ levels in Section 3.3.2, before proceeding to the general case in Section 3.3.3. Finally, the algorithm to compute the optimal subset of levels is described in Section 3.3.4.

3.3.1 Assumptions

In this chapter, we assume that failures from different levels are independent[1]. For each level ℓ, the arrival of failures follows *Poisson* process with error rate λ_ℓ. In order to deal with the interplay of failures from different levels, we make use of the following well-known properties of independent Poisson processes [55, Chapter 2.3].

Property 1. *During the execution of a segment with length w, let X_ℓ denote the time when the first level-ℓ error strikes. Thus, X_ℓ is a random variable following an Exponential distribution*

[1] In practice, failures from different checkpointing levels can exhibit potential correlation [37, 61]. Consideration of correlated failures is beyond the scope of this work.

with parameter λ_ℓ, for all $\ell = 1, 2, \ldots, k$.

(1). Let X denote the time when the first error (of any level) strikes. We have $X = \min\{X_1, X_2, \ldots, X_k\}$, which follows an Exponential distribution with parameter $\Lambda = \sum_{\ell=1}^{k} \lambda_\ell$. The probability of having an error (from any level) in the segment is therefore $P(X \leq w) = 1 - e^{-\Lambda w}$.

(2). Given that an error (from any level) strikes during the execution of the segment, the probability that the error belongs to a particular level is proportional to the error rate of that level, i.e., $P(X = X_\ell | X \leq w) = \frac{\lambda_\ell}{\Lambda}$, for all $\ell = 1, 2, \ldots, k$.

Moreover, we assume that error rates of different levels are of the same order, i.e., $\lambda_\ell = \Theta(\Lambda)$ for all $\ell = 1, 2, \ldots, k$, and that errors only strike during the computations, while checkpointing and recovery are error-free. Indeed, the durations of checkpoints and recoveries are generally small compared to the pattern length, so the probability of a failure striking during these operations is low. It has been shown in Chapter 2 that removing this assumption does not impact the first-order approximation of the pattern overhead.

3.3.2 Optimal two-level pattern

We start by analyzing the two-level pattern shown in Figure 3.2(b). The goal is to determine a first-order approximation to the optimal pattern length W, the number n of level-1 checkpoints in the pattern, as well as the length $w_i = \alpha_i W$ of the i-th segment, for all $1 \leq i \leq n$, where $\sum_{i=1}^{n} \alpha_i = 1$.

With a single segment

We first consider a special case of the two-level pattern, in which only a single segment is present, i.e., $n = 1$. The result establishes the order of the optimal pattern length W^{opt}, which will be used later for analyzing the general case. Recall that $\Lambda = \lambda_1 + \lambda_2$ and, for convenience, let us also define $C = C_1 + C_2$. The following proposition shows the expected time of such a pattern with fixed length W.

Proposition 6. *The expected execution time of a two-level pattern with a single segment and fixed length W is*

$$\mathbb{E} = W + C + \frac{1}{2}\Lambda W^2 + O(\max\{\Lambda^2 W^3, \Lambda W\}).$$

Proof. We can express the expected execution time of the pattern recursively as follows:

$$\begin{aligned}\mathbb{E} = P\left(\mathbb{E}^{\text{lost}}(W, \Lambda) + \frac{\lambda_1}{\Lambda}\left(R_1 + \mathbb{E}\right) + \frac{\lambda_2}{\Lambda}\Big(R_2 + R_1 + \mathbb{E}\Big)\right) \\ + (1 - P)\left(W + C\right), \end{aligned} \tag{3.3}$$

where $P = 1 - e^{-\Lambda W}$ denotes the probability of having a failure (either level-1 or level-2) during the execution of the pattern based on Property 1.1, and $\mathbb{E}^{\text{lost}}(w_i, \Lambda)$ denotes the expected

time lost when such a failure occurs. In this case, and based on Property 1.2, if the failure belongs to level 1, which happens with probability $\frac{\lambda_1}{\Lambda}$, we can recover from the latest level-1 checkpoint (R_1). Otherwise, the failure belongs to level 2 with probability $\frac{\lambda_2}{\Lambda}$, and we need to first recover from the latest level-2 checkpoint (R_2) before restoring the level-1 checkpoint (R_1). In both cases, the entire pattern needs to be re-executed again. Finally, if no error (of any level) strikes, which happens with probability $1-P$, the pattern is completed after W time of execution followed by the time C to perform the two checkpoints, which are assumed to be error-free.

From [62, Equation (1.13)], the expected time lost when executing a segment of length W with error rate Λ is

$$\mathbb{E}^{\text{lost}}(W,\Lambda) = \frac{1}{\Lambda} - \frac{W}{e^{\Lambda W}-1}. \tag{3.4}$$

Substituting Equation (3.4) into Equation (3.3) and solving for $\mathbb{E}$, we get:

$$\mathbb{E} = \left(e^{\Lambda W}-1\right)\left(\frac{1}{\Lambda} + R_1 + \frac{\lambda_2}{\Lambda}R_2\right) + C_1 + C_2, \tag{3.5}$$

which is an exact formula on the expected execution time of the pattern. Now, using Taylor series to expand $e^{\Lambda W} = 1 + \Lambda W + \frac{\Lambda^2 W^2}{2} + O(\Lambda^3 W^3)$ while assuming $W = \Theta(\Lambda^{-x})$, where $0 < x < 1$, we can re-write Equation (3.5) as

$$\begin{aligned}\mathbb{E} &= W + \frac{1}{2}\Lambda W^2 + C_1 + C_2 + O(\Lambda^2 W^3) \\ &\quad + \left(\Lambda W + \frac{\Lambda^2 W^2}{2} + O(\Lambda^3 W^3)\right)\left(R_1 + \frac{\lambda_2}{\Lambda}R_2\right).\end{aligned}$$

Since recovery costs (R_1, R_2) are assumed to be constants, and error rates ($\lambda_1, \lambda_2, \Lambda$) are in the same order, the expected execution time can be expressed as follows:

$$\mathbb{E} = W + C_1 + C_2 + \frac{1}{2}\Lambda W^2 + O(\Lambda^2 W^3) + O(\Lambda W),$$

which completes the proof of the proposition. □

From Proposition 6, the expected execution overhead of the pattern can be derived as

$$H = \frac{C}{W} + \frac{1}{2}\Lambda W + O(\max\{\Lambda^2 W^2, \Lambda\}).$$

Assume that the platform MTBF $\mu = 1/\Lambda$ is large in front of the resilience parameters, and consider the first two terms of H: the overhead is minimized when the pattern has length $W = \Theta(\Lambda^{-1/2})$, and in that case both terms are in the order of $\Theta(\Lambda^{1/2})$, so we have $H = \Theta(\Lambda^{1/2}) + O(\Lambda)$. Indeed, the last term $O(\Lambda^2 W^2) = O(\Lambda)$ becomes negligible compared to $\Theta(\Lambda^{1/2})$. Hence, the optimal pattern length W^{opt} can be obtained by balancing the first two terms in H, which gives

$$W^{\text{opt}} = \sqrt{\frac{2C}{\Lambda}} = \Theta(\Lambda^{-1/2}), \tag{3.6}$$

and the optimal execution overhead becomes

$$H^{\text{opt}} = \sqrt{2\Lambda C} + O(\Lambda). \tag{3.7}$$

Remarks. Unlike in single-level checkpointing, the checkpoint to roll back to in a two-level pattern depends on which type of error strikes first. Under first-order approximation and assuming that the resilience parameters are small compared to the platform MTBF and pattern length, the formulas shown in Equations (3.6) and (3.7) reduce exactly to Young/Daly's classical result by aggregating the error rates and checkpointing costs of both levels.

With multiple segments

We now consider the general two-level pattern with multiple segments, and derive the optimal pattern parameters. As in the single-segment case, we start with a proposition showing the expected time to execute a two-level pattern with fixed parameters.

Proposition 7. *The expected execution time of a given two-level pattern is*

$$\mathbb{E} = W + nC_1 + C_2 + \frac{1}{2}\left(\lambda_1 \sum_{i=1}^{n} \alpha_i^2 + \lambda_2\right) W^2 + O(\Lambda^{1/2}).$$

Proof. We first prove the following result (by induction) on the expected time $\mathbb{E}_i$ to execute the i-th segment of the pattern (up to the level-1 checkpoint at the end of the segment):

$$\mathbb{E}_i = w_i + C_1 + \frac{\lambda_1}{2} w_i^2 + \lambda_2 \left(\frac{w_i^2}{2} + \sum_{j=1}^{i-1} w_j w_i \right) + O(\Lambda^{1/2}). \tag{3.8}$$

According to the result with a single segment, we know that the optimal pattern length and hence the segment length are in the order of $O(\Lambda^{-1/2})$, which implies that $\mathbb{E}_i = w_i + O(1)$.

For the ease of analysis, we assume that there is a hypothetical segment at the beginning of the pattern with length $w_0 = 0$ (hence no need to checkpoint). For this segment, we have $\mathbb{E}_0 = w_0 = 0$, satisfying Equation (3.8). Suppose the claim holds up to $\mathbb{E}_{i-1}$. Then, $\mathbb{E}_i$ can be recursively expressed as follows:

$$\begin{aligned} \mathbb{E}_i = P_i \Bigg(& \mathbb{E}^{\text{lost}}(w_i, \Lambda) + \frac{\lambda_1}{\Lambda} (R_1 + \mathbb{E}_i) \\ & + \frac{\lambda_2}{\Lambda} \Big(R_2 + R_1 + \sum_{j=1}^{i-1} \mathbb{E}_j + \mathbb{E}_i \Big) \Bigg) \\ & + (1 - P_i)(w_i + C_1), \end{aligned} \tag{3.9}$$

where $P_i = 1 - e^{-\Lambda w_i}$ denotes the probability of having a failure (either level-1 or level-2) during the execution of the segment, and $\mathbb{E}^{\text{lost}}(w_i, \Lambda)$ denotes the expected time lost when such a failure occurs.

Equation (3.9) is very similar to Equation (3.3), except when a level-2 failure occurs we need to re-execute all the segments (up to segment i) that have been executed so far. Following the derivation of Proposition 6 and applying $\mathbb{E}_j = w_j + O(1)$ for $j = 1, 2, \ldots, i-1$, we can derive the first-order approximation of $\mathbb{E}_i$ as follows:

$$\begin{aligned}\mathbb{E}_i &= w_i + C_1 + \frac{1}{2}\left(\lambda_1 w_i^2 + \lambda_2 w_i^2 + 2\lambda_2 w_i \sum_{j=1}^{i-1} \mathbb{E}_j\right) + O(\Lambda^{1/2}) \\ &= w_i + C_1 + \frac{1}{2}\left(\lambda_1 w_i^2 + \lambda_2 w_i^2 + 2\lambda_2 w_i \sum_{j=1}^{i-1}\Big(w_j + O(1)\Big)\right) + O(\Lambda^{1/2}) \\ &= w_i + C_1 + \frac{1}{2}\left(\lambda_1 w_i^2 + \lambda_2\Big(w_i^2 + 2\sum_{j=1}^{i-1} w_j w_i\Big)\right) + O(\Lambda^{1/2}). \end{aligned} \tag{3.10}$$

Since the level-2 checkpoint at the end of the pattern is also assumed to be error-free, we can compute the expected execution time of the pattern as

$$\begin{aligned}\mathbb{E} &= \sum_{i=1}^{n} \mathbb{E}_i + C_2 \\ &= W + nC_1 + C_2 + \frac{1}{2}\left(\lambda_1 \sum_{i=1}^{n} \alpha_i^2 + \lambda_2\right) W^2 + O(\Lambda^{1/2}),\end{aligned}$$

since $\sum_{i=1}^{n} w_i^2 + 2\sum_{i=1}^{n}\sum_{j=1}^{i-1} w_j w_i = \left(\sum_{i=1}^{n} w_i\right)^2 = W^2$. □

Theorem 9. *A first-order approximation to the optimal two-level pattern is characterized by*

$$n^{opt} = \sqrt{\frac{\lambda_1}{\lambda_2} \cdot \frac{C_2}{C_1}}, \tag{3.11}$$

$$\alpha_i^{opt} = \frac{1}{n^{opt}} \quad \forall i = 1, 2, \ldots, n^{opt}, \tag{3.12}$$

$$W^{opt} = \sqrt{\frac{n^{opt} C_1 + C_2}{\frac{1}{2}\left(\frac{\lambda_1}{n^{opt}} + \lambda_2\right)}}, \tag{3.13}$$

where n^{opt} is the number of segments, $\alpha_i^{opt} W^{opt}$ is the length of the i-th segment, and W^{opt} is the pattern length.
The optimal pattern overhead is

$$H^{opt} = \sqrt{2\lambda_1 C_1} + \sqrt{2\lambda_2 C_2} + O(\Lambda). \tag{3.14}$$

Proof. For a given pattern with a fixed number n of segments, $\sum_{i=1}^{n} \alpha_i^2$ is minimized subject to $\sum_{i=1}^{n} \alpha_i = 1$ when $\alpha_i = \frac{1}{n}$ for all $i = 1, 2, \ldots, n$. Hence, we can derive the expected execution overhead from Proposition 7 as follows:

$$H = \frac{nC_1 + C_2}{W} + \frac{1}{2}\left(\frac{\lambda_1}{n} + \lambda_2\right) W + O(\Lambda). \tag{3.15}$$

For a given n, the optimal work length can then be computed from Equation (3.15), and it is given by $W^{\text{opt}} = \sqrt{\frac{nC_1+C_2}{\frac{1}{2}\left(\frac{\lambda_1}{n}+\lambda_2\right)}}$. In that case, the execution overhead becomes

$$H = \sqrt{2\left(\frac{\lambda_1}{n}+\lambda_2\right)(nC_1+C_2)} + O(\Lambda), \tag{3.16}$$

which is minimized as shown in Equation (3.14) when n satisfies Equation (3.11). Indeed, $2\left(\frac{\lambda_1}{n^{\text{opt}}}+\lambda_2\right)(n^{\text{opt}}C_1+C_2) = 2\lambda_1C_1 + 2\lambda_2C_2 + 4\sqrt{\lambda_1\lambda_2C_1C_2} = (\sqrt{2\lambda_1C_1}+\sqrt{2\lambda_1C_1})^2$. In practice, since the number of segments can only be a positive integer, the optimal solution is either $\max(1, \lfloor n^{\text{opt}} \rfloor)$ or $\lceil n^{\text{opt}} \rceil$, whichever leads to a smaller value of the convex function H as shown in Equation (3.16). □

Remarks. Consider the example given in [39] with $C_1 = R_1 = 20$, $C_2 = R_2 = 50$, $\lambda_1 = 2.78 \times 10^{-4}$ and $\lambda_2 = 4.63 \times 10^{-5}$. The optimal solution[2] provided by [39] gives $n^{\text{opt}} = 3.83$, $W^{\text{opt}} = 1362.49$ and $H^{\text{opt}} = 0.1879$, while Theorem 9 suggests $n^{\text{opt}} = 3.87$, $W^{\text{opt}} = 1378.27$ and $H^{\text{opt}} = 0.1735$, which is quite close to the exact optimum. The difference in overhead is due to the negligence of lower-order terms in the first-order approximation. We point out that the solution provided by [39] relies on numerical methods to solve rather complex mathematical equations, whose convergence is not always guaranteed, and it is only applicable to two levels. Our result, on the other hand, is able to provide fast and good approximation to the optimal solution when the error rates are sufficiently small, and it can be readily extended to an arbitrary number of levels, as shown in the next section.

3.3.3 Optimal k-level pattern

In this section, we derive the first-order approximation to the optimal k-level pattern by determining its length W, the number N_ℓ of level-ℓ checkpoints for all $1 \leq \ell \leq k$, as well as the positions of all checkpoints in the pattern.

Observations

Before analyzing the optimal pattern, we make several observations. First, we can obtain the orders of the optimal length and pattern overhead as shown below (recall that $\Lambda = \sum_{\ell=1}^{k} \lambda_\ell$).

Observation 1. *Consider the simplest k-level pattern with a single segment of length W. We can conduct the same analysis as in Section 3.3.2 to show that the optimal pattern length satisfies $W^{opt} = \Theta(\Lambda^{-1/2})$, and the corresponding overhead satisfies $H^{opt} = \Theta(\Lambda^{1/2})$.*

From the analysis of the two-level pattern, we can also observe that the overall execution overhead of any pattern comes from two distinct sources defined below.

Observation 2. *There are two types of execution overheads for a pattern:*

[2]The original optimal solution of [39] considers faults in checkpointing but not during recoveries. We adapt its solution to exclude faults in checkpointing so to be consistent with the model in this chapter for a fair comparison. The results reported herein are based on this modified solution.

(1). Error-free overhead*, denoted as o_{ef}, is the total cost of all the checkpoints placed in the pattern. For a given set of checkpoints, the error-free overhead is completely determined regardless of their positions in the pattern.*

(2). Re-executed fraction overhead*, denoted as o_{re}, is the expected fraction of work that needs to be re-executed due to errors. The re-executed fraction overhead depends on both the set of checkpoints and their positions.*

For example, in the two-level pattern with n level-1 checkpoints and given values of α_i for all $i = 1, 2, \ldots, n$, the two types of overheads are given by $o_{\text{ef}} = nC_1 + C_2$ and $o_{\text{re}} = \frac{1}{2}\left(f_1 \sum_{i=1}^{n} \alpha_i^2 + f_2\right)$, where $f_\ell = \frac{\lambda_\ell}{\Lambda}$ for $\ell = 1, 2$. Assuming that checkpoints at all levels have constant costs and that the error rates at all levels are in the same order, then both o_{ef} and o_{re} can be considered as constants, i.e., $o_{\text{ef}} = O(1)$ and $o_{\text{re}} = O(1)$.

A trade-off exists between these two types of execution overheads, since placing more checkpoints generally reduces the re-executed work fraction when an error strikes, but it can adversely increase the overhead when the execution is error-free. Therefore, in order to achieve the best overall overhead, a resilience algorithm must seek an optimal balance between o_{ef} and o_{re}.

For a given pattern with fixed overheads o_{ef} and o_{re}, we can make the following observation based on Propositions 6 and 7, which partially characterizes the optimal pattern.

Observation 3. *For a given pattern (with fixed o_{ef} and o_{re}), the expected execution time is given by*

$$\mathbb{E} = \underbrace{W + o_{\text{ef}}}_{\substack{\textit{error-free}\\ \textit{execution time}}} + \underbrace{\Lambda W}_{\substack{\textit{expected}\\ \textit{\# errors}}} \cdot \underbrace{o_{\text{re}} W}_{\substack{\textit{re-executed work}\\ \textit{in case of error}}} + O(\Lambda^{1/2}), \tag{3.17}$$

and the optimal pattern length and the resulting expected execution overhead of the pattern are

$$W^{opt} = \sqrt{\frac{o_{\text{ef}}}{\Lambda \cdot o_{\text{re}}}}, \tag{3.18}$$

$$H^{opt} = 2\sqrt{\Lambda \cdot o_{\text{ef}} \cdot o_{\text{re}}} + O(\Lambda). \tag{3.19}$$

Equation (3.19) shows that the trade-off between o_{ef} and o_{re} is manifested as the product of the two terms. Hence, in order to determine the optimal pattern, it suffices to find the pattern parameters (e.g., n and α_i) that minimize $o_{\text{ef}} \cdot o_{\text{re}}$.

Analysis

We now extend the analysis to derive the optimal multi-level checkpointing patterns. Generally, for a k-level pattern, each computational segment $s^{(\ell)}_{i_{k-1},\ldots,i_\ell}$ can be uniquely identified by its level ℓ as well as its position $\langle i_{k-1}, \ldots, i_\ell \rangle$ within the multi-level hierarchy. For instance, in a four-level pattern, the segment $s^{(2)}_{1,3}$ denotes the third level-2 segment inside the first level-3 segment of the pattern (see Figure 3.3). Note that a segment can contain multiple sub-segments at the lower levels (except for bottom-level segments) and is a sub-segment of a larger segment at a

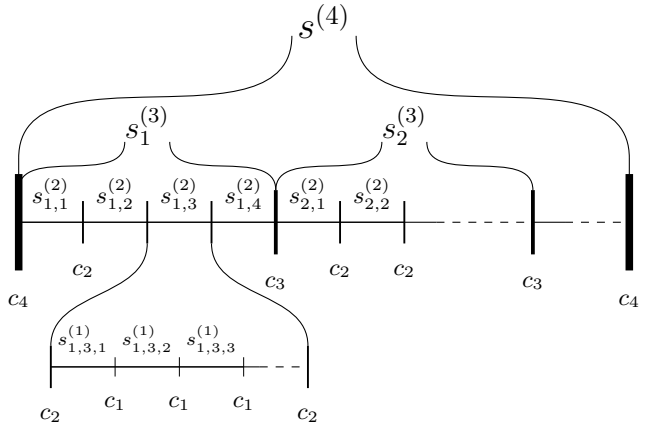


Figure 3.3: Example of a 4-level pattern. Here, we let $c_\ell = C_1|C_2|\cdots|C_\ell$ denote the succession of checkpoints from level 1 to level ℓ.

higher level (except for top-level segments). The entire pattern can be denoted as $s^{(k)}$, which is the only segment at level k.

For any segment $s^{(\ell)}_{i_{k-1},\ldots,i_\ell}$ at level ℓ, where $1 \le \ell \le k$, let $w^{(\ell)}_{i_{k-1},\ldots,i_\ell}$ denote its length. Hence, we have $w^{(\ell+1)}_{i_{k-1},\ldots,i_{\ell+1}} = \sum_{i_\ell} w^{(\ell)}_{i_{k-1},\ldots,i_\ell}$ and $w^{(k)} = W$. Also, let $n^{(\ell)}_{i_{k-1},\ldots,i_\ell}$ denote the number of sub-segments contained by $s^{(\ell)}_{i_{k-1},\ldots,i_\ell}$ at the lower level $\ell - 1$. We have $n^{(1)}_{i_{k-1},\ldots,i_1} = 1$ for all $i_{k-1},\ldots,i_1$. For convenience, we further define

$$\alpha^{(\ell)}_{i_{k-1},\ldots,i_\ell} = \frac{w^{(\ell)}_{i_{k-1},\ldots,i_\ell}}{W}$$

as the fraction of the length of segment $s^{(\ell)}_{i_{k-1},\ldots,i_\ell}$ inside the pattern, and define N_ℓ to be the total number of level-ℓ segments in the entire pattern. Therefore, we have $N_k = 1$, $N_{k-1} = n^{(k)}$, and in general

$$N_\ell = \sum_{i_{k-1},\ldots,i_{\ell+1}} n^{(\ell+1)}_{i_{k-1},\ldots,i_{\ell+1}}.$$

The following proposition shows the expected time to execute a given k-level pattern.

Proposition 8. *The expected execution time of a given k-level pattern is*

$$\mathbb{E} = W + \sum_{\ell=1}^{k-1} N_\ell C_\ell + C_k + \frac{W^2}{2}\left(\sum_{\ell=1}^{k}\lambda_\ell \sum_{i_{k-1},\ldots,i_\ell}\left(\alpha^{(\ell)}_{i_{k-1},\ldots,i_\ell}\right)^2\right) + O(\Lambda^{1/2}).$$

Proof. We show that the expected time to execute any segment $s^{(h)}_{i_{k-1},\ldots,i_h}$ at level h, where $1 \le h \le k$, satisfies the following (without counting the time to execute all the checkpoints

inside the segment):

$$\begin{aligned}\mathbb{E}^{(h)}_{i_{k-1},\dots,i_h} &= w^{(h)}_{i_{k-1},\dots,i_h} + \frac{W^2}{2}\left(\sum_{\ell=1}^{h}\lambda_\ell \sum_{i_{h-1},\dots,i_\ell}\left(\alpha^{(\ell)}_{i_{k-1},\dots,i_\ell}\right)^2\right)\\ &\quad + \Lambda_{[h+1,k]}\left(\frac{\left(w^{(h)}_{i_{k-1},\dots,i_h}\right)^2}{2} + w^{(h)}_{i_{k-1},\dots,i_h}\sum_{j_h=1}^{i_h-1}\mathbb{E}^{(h)}_{i_{k-1},\dots,j_h}\right)\\ &\quad + w^{(h)}_{i_{k-1},\dots,i_h}\sum_{\ell=h+2}^{k}\Lambda_{[\ell,k]}\sum_{j_{\ell-1}=1}^{i_{\ell-1}-1}\mathbb{E}^{(\ell-1)}_{i_{k-1},\dots,j_{\ell-1}} + O(\Lambda^{1/2}),\end{aligned} \tag{3.20}$$

where $\Lambda_{[x,y]} = \sum_{\ell=x}^{y}\lambda_\ell$ and, if $x > y$, we define $\Lambda_{[x,y]} = 0$. The proposition can then be proven by setting $\mathbb{E} = \mathbb{E}^{(k)} + \sum_{\ell=1}^{k-1} N_\ell C_\ell + C_k$, since checkpoints are assumed to be error-free.

We now prove Equation (3.20) by induction on the level h. For the base case, i.e., when $h = 1$, consider a segment $s^{(1)}_{i_{k-1},\dots,i_1}$ at the first level. Following the proof of Proposition 7 (in particular, Equation (3.9)), we can express its expected execution time $\mathbb{E}^{(1)}_{i_{k-1},\dots,i_1}$, as

$$\begin{aligned}\mathbb{E}^{(1)}_{i_{k-1},\dots,i_1} = P^{(1)}_{i_{k-1},\dots,i_1}\Bigg(&\mathbb{E}^{\text{lost}}\left(w^{(1)}_{i_{k-1},\dots,i_1},\Lambda\right)\\ &+\frac{\lambda_1}{\Lambda}\left(R_1 + \mathbb{E}^{(1)}_{i_{k-1},\dots,i_1}\right)\\ &+\frac{\lambda_2}{\Lambda}\left(\sum_{j=1}^{2}R_j + \sum_{j_1=1}^{i_1}\mathbb{E}^{(1)}_{i_{k-1},\dots,j_1}\right)\\ &+\frac{\lambda_3}{\Lambda}\left(\sum_{j=1}^{3}R_j + \sum_{j_2=1}^{i_2-1}\mathbb{E}^{(2)}_{i_{k-1},\dots,j_2} + \sum_{j_1=1}^{i_1}\mathbb{E}^{(1)}_{i_{k-1},\dots,j_1}\right)\\ &\quad\vdots\\ &+\frac{\lambda_k}{\Lambda}\left(\sum_{j=1}^{k}R_j + \sum_{j_{k-1}=1}^{i_{k-1}-1}\mathbb{E}^{(k-1)}_{j_{k-1}} + \sum_{j_{k-2}=1}^{i_{k-2}-1}\mathbb{E}^{(k-2)}_{i_{k-1},j_{k-2}} + \cdots + \sum_{j_1=1}^{i_1}\mathbb{E}^{(1)}_{i_{k-1},\dots,j_1}\right)\Bigg)\\ &+\left(1 - P^{(1)}_{i_{k-1},\dots,i_1}\right)w^{(1)}_{i_{k-1},\dots,i_1},\end{aligned} \tag{3.21}$$

where $\Lambda = \sum_{\ell=1}^{k}\lambda_\ell$ is the total rate of all error sources, and $P^{(1)}_{i_{k-1},\dots,i_1} = 1 - e^{\Lambda\cdot w^{(1)}_{i_{k-1},\dots,i_1}}$ denotes the probability of having an error (from any level) during the execution of the segment.

Simplifying Equation (3.21) and solving for $\mathbb{E}^{(1)}_{i_{k-1},\dots,i_1}$ we get:

$$\begin{aligned}
\mathbb{E}^{(1)}_{i_{k-1},\dots,i_1} &= w^{(1)}_{i_{k-1},\dots,i_1} + \frac{W^2}{2}\Lambda_{[1,k]}\left(\alpha^{(1)}_{i_{k-1},\dots,i_1}\right)^2 \\
&\quad + w^{(1)}_{i_{k-1},\dots,i_1}\sum_{\ell=2}^{k}\Lambda_{[\ell,k]}\sum_{j_{\ell-1}=1}^{i_{\ell-1}-1}\mathbb{E}^{(\ell-1)}_{i_{k-1},\dots,j_{\ell-1}} + O(\Lambda^{1/2}) \\
&= w^{(1)}_{i_{k-1},\dots,i_1} + \frac{W^2}{2}\lambda_1\left(\alpha^{(1)}_{i_{k-1},\dots,i_1}\right)^2 \\
&\quad + \Lambda_{[2,k]}\left(\frac{\left(w^{(1)}_{i_{k-1},\dots,i_1}\right)^2}{2} + w^{(1)}_{i_{k-1},\dots,i_1}\sum_{j_1=1}^{i_1-1}\mathbb{E}^{(1)}_{i_{k-1},\dots,j_1}\right) \\
&\quad + w^{(1)}_{i_{k-1},\dots,i_1}\sum_{\ell=3}^{k}\Lambda_{[\ell,k]}\sum_{j_{\ell-1}=1}^{i_{\ell-1}-1}\mathbb{E}^{(\ell-1)}_{i_{k-1},\dots,j_{\ell-1}} + O(\Lambda^{1/2}),
\end{aligned}$$

which satisfies Equation (3.20).

Suppose Equation (3.20) holds up to any segment $s^{(h)}_{i_{k-1},\dots,i_h}$ at level h. Following the proof of Proposition 7 (in particular, the derivation of Equation (3.10)), we can show by induction that $\mathbb{E}^{(h)}_{i_{k-1},\dots,i_h} = w^{(h)}_{i_{k-1},\dots,i_h} + O(1)$. Hence, for segment $s^{(h+1)}_{i_{k-1},\dots,i_{h+1}}$ at level $h+1$, we have:

$$\begin{aligned}
\mathbb{E}^{(h+1)}_{i_{k-1},\dots,i_{h+1}} &= \sum_{i_h}\mathbb{E}^{(h)}_{i_{k-1},\dots,i_h} \\
&= \sum_{i_h} w^{(h)}_{i_{k-1},\dots,i_h} + \frac{W^2}{2}\left(\sum_{\ell=1}^{h}\lambda_\ell\sum_{i_h,\dots,i_\ell}\left(\alpha^{(\ell)}_{i_{k-1},\dots,i_\ell}\right)^2\right) \\
&\quad + \Lambda_{[h+1,k]}\sum_{i_h}\left(\frac{\left(w^{(h)}_{i_{k-1},\dots,i_h}\right)^2}{2} + w^{(h)}_{i_{k-1},\dots,i_h}\sum_{j_h=1}^{i_h-1} w^{(h)}_{i_{k-1},\dots,j_h}\right) \\
&\quad + \sum_{i_h} w^{(h)}_{i_{k-1},\dots,i_h}\sum_{\ell=h+2}^{k}\Lambda_{[\ell,k]}\sum_{j_{\ell-1}=1}^{i_{\ell-1}-1}\mathbb{E}^{(\ell-1)}_{i_{k-1},\dots,j_{\ell-1}} + O(\Lambda^{1/2}) \\
&= w^{(h+1)}_{i_{k-1},\dots,i_{h+1}} + \frac{W^2}{2}\left(\sum_{\ell=1}^{h}\lambda_\ell\sum_{i_h,\dots,i_\ell}\left(\alpha^{(\ell)}_{i_{k-1},\dots,i_\ell}\right)^2\right) + \Lambda_{[h+1,k]}\frac{\left(w^{(h+1)}_{i_{k-1},\dots,i_{h+1}}\right)^2}{2} \\
&\quad + w^{(h+1)}_{i_{k-1},\dots,i_{h+1}}\sum_{\ell=h+2}^{k}\Lambda_{[\ell,k]}\sum_{j_{\ell-1}=1}^{i_{\ell-1}-1}\mathbb{E}^{(\ell-1)}_{i_{k-1},\dots,j_{\ell-1}} + O(\Lambda^{1/2})
\end{aligned}$$

$$= w^{(h+1)}_{i_{k-1},\ldots,i_{h+1}} + \frac{W^2}{2}\left(\sum_{\ell=1}^{h+1}\lambda_\ell \sum_{i_h,\ldots,i_\ell}\left(\alpha^{(\ell)}_{i_{k-1},\ldots,i_\ell}\right)^2\right)$$
$$+ \Lambda_{[h+2,k]}\left(\frac{\left(w^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)^2}{2} + w^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\sum_{j_{h+1}=1}^{i_{h+1}-1}\mathbb{E}^{(h+1)}_{i_{k-1},\ldots,j_{h+1}}\right)$$
$$+ w^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\sum_{\ell=h+3}^{k}\Lambda_{[\ell,k]}\sum_{j_{\ell-1}=1}^{i_{\ell-1}-1}\mathbb{E}^{(\ell-1)}_{i_{k-1},\ldots,j_{\ell-1}} + O(\Lambda^{1/2}).$$

Hence, Equation (3.20) also holds for any segment at level $h+1$. This completes the proof of the proposition. □

Proposition 8 shows that, for a given k-level checkpointing pattern, the error-free overhead o_{ef} and the re-executed fraction overhead o_{re} are given as follows:

$$o_{\text{ef}} = \sum_{\ell=1}^{k-1} N_\ell C_\ell + C_k, \tag{3.22}$$

$$o_{\text{re}} = \frac{1}{2}\sum_{\ell=1}^{k} f_\ell \sum_{i_{k-1},\ldots,i_\ell}\left(\alpha^{(\ell)}_{i_{k-1},\ldots,i_\ell}\right)^2, \tag{3.23}$$

where $f_\ell = \frac{\lambda_\ell}{\Lambda}$. According to Observation 3, it remains to find parameters of the pattern such that $o_{\text{ef}} \cdot o_{\text{re}}$ is minimized.

To derive the optimal pattern, we first consider the case where o_{ef} is fixed, i.e., the set of checkpoints is given. The following proposition shows the optimal value of o_{re}.

Proposition 9. *For a k-level checkpointing pattern, suppose the number N_ℓ of checkpoints at each level ℓ is given, i.e., the error-free overhead o_{ef} is fixed (as in Equation (3.22)). Then, the optimal value of the re-executed work overhead is given by*

$$o^{opt}_{\text{re}} = \frac{1}{2}\left(\sum_{\ell=1}^{k-1}\frac{f_\ell}{N_\ell} + f_k\right), \tag{3.24}$$

and it is obtained when all the checkpoints of each level are equally spaced in the pattern.

Proof. According to Equation (3.23), which shows the value of o_{re} for the entire pattern, we can define the corresponding overhead for each level-h segment $s^{(h)}_{i_{k-1},\ldots,i_h}$ recursively as follows:

$$o_{\text{re}}\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right) = \frac{f_h}{2}\cdot\left(\alpha^{(h)}_{i_{k-1},\ldots,i_h}\right)^2 + \sum_{i_{h-1}} o_{\text{re}}\left(s^{(h-1)}_{i_{k-1},\ldots,i_{h-1}}\right),$$

with $o_{\text{re}}\left(s^{(0)}_{i_{k-1},\ldots,i_0}\right) = 0$ by definition.

For each segment $s^{(h)}_{i_{k-1},\ldots,i_h}$, we also define $N_\ell\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right)$ to be the total number of level-ℓ segments it contains, with $\ell \leq h$. We will show that the optimal value $o^{\text{opt}}_{\text{re}}\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right)$ for the segment satisfies:

$$o^{\text{opt}}_{\text{re}}\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right) = \frac{1}{2}\left(\sum_{\ell=1}^{h} \frac{f_\ell}{N_\ell\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right)}\right)\left(\alpha^{(h)}_{i_{k-1},\ldots,i_h}\right)^2, \tag{3.25}$$

and it is achieved when its level-ℓ checkpoints are equally spaced, for all $\ell \leq h-1$. The proposition can then be proven by setting $o^{\text{opt}}_{\text{re}} = o^{\text{opt}}_{\text{re}}\left(s^{(k)}\right)$, since $N_\ell\left(s^{(k)}\right) = N_\ell$, $N_k = 1$, and $\alpha^{(k)} = 1$.

Now, we prove Equation (3.25) by induction on the level h. For the base case, i.e., when $h = 1$, we have $o_{\text{re}}\left(s^{(1)}_{i_{k-1},\ldots,i_1}\right) = \frac{f_1}{2} \cdot \left(\alpha^{(1)}_{i_{k-1},\ldots,i_1}\right)^2$ by definition, and it satisfies Equation (3.25), because $N_1\left(s^{(1)}_{i_{k-1},\ldots,i_1}\right) = 1$. Suppose Equation (3.25) holds for any segment $s^{(h)}_{i_{k-1},\ldots,i_h}$ at level h. Then, for segment $s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}$ at level $h+1$, we have:

$$\begin{aligned} o_{\text{re}}\left(s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right) &= \frac{f_{h+1}}{2} \cdot \left(\alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)^2 + \sum_{i_h} o^{\text{opt}}_{\text{re}}\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right) \\ &= \frac{f_{h+1}}{2} \cdot \left(\alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)^2 + \frac{1}{2}y, \end{aligned} \tag{3.26}$$

where $y = \sum_{i_h} x^{(h)}_{i_{k-1},\ldots,i_h} \cdot \left(\alpha^{(h)}_{i_{k-1},\ldots,i_h}\right)^2$, and $x^{(h)}_{i_{k-1},\ldots,i_h} = \sum_{\ell=1}^{h} \frac{f_\ell}{N_\ell\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right)}$. To minimize $o_{\text{re}}\left(s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)$ as shown in Equation (3.26), it suffices to solve the following minimization problem:

$$\begin{aligned} &\text{minimize} \quad y = \sum_{i_h} x^{(h)}_{i_{k-1},\ldots,i_h} \cdot \left(\alpha^{(h)}_{i_{k-1},\ldots,i_h}\right)^2, \\ &\text{subject to} \quad \sum_{i_h} \alpha^{(h)}_{i_{k-1},\ldots,i_h} = \alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}. \end{aligned}$$

Since y is clearly a convex function of $\alpha^{(h)}_{i_{k-1},\ldots,i_h}$, we can readily get, using Lagrange multiplier [18], the minimum value of y as follows:

$$y_{\min} = \frac{1}{\sum_{i_h} 1/x^{(h)}_{i_{k-1},\ldots,i_h}} \cdot \left(\alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)^2, \tag{3.27}$$

which is obtained at

$$\widetilde{\alpha}^{(h)}_{i_{k-1},\ldots,i_h} = \frac{1/x^{(h)}_{i_{k-1},\ldots,i_h}}{\sum_{j_h} 1/x^{(h)}_{i_{k-1},\ldots,j_h}} \cdot \alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}. \tag{3.28}$$

Let us define $z = \sum_{i_h} 1/x^{(h)}_{i_{k-1},\ldots,i_h}$. We now need to solve the following maximization problem:

$$\begin{aligned}\text{maximize } z &= \sum_{i_h} \frac{1}{\sum_{\ell=1}^{h} \frac{f_\ell}{N_\ell\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right)}},\\ \text{subject to } &\sum_{i_h} N_\ell\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right) = N_\ell\left(s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right), \forall \ell = 1,\ldots,h.\end{aligned}$$

Again, z is a convex function of $N_\ell\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right)$, and it can be shown to be maximized when

$$N_\ell\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right) = \frac{N_\ell\left(s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)}{n^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}}, \quad \forall \ell = 1,\ldots,h,$$

which gives $\widetilde{\alpha}^{(h)}_{i_{k-1},\ldots,i_h} = \frac{1}{n^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}} \cdot \alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}$ according to Equation (3.28). This implies that all level-ℓ checkpoints are also equally spaced inside segment $s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}$, for all $\ell \leq h$. The maximum value of z in this case is

$$z_{\max} = \frac{1}{\sum_{\ell=1}^{h} \frac{f_\ell}{N_\ell\left(s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)}},$$

and the optimal value of $y_{\min}$ according to Equation (3.27) is then given by

$$\begin{aligned}y^{\text{opt}}_{\min} &= \frac{1}{z_{\max}} \left(\alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)^2\\ &= \left(\sum_{\ell=1}^{h} \frac{f_\ell}{N_\ell\left(s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)}\right) \left(\alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)^2.\end{aligned}$$

Substituting $y^{\text{opt}}_{\min}$ into Equation (3.26), we get the optimal value of $o_{\text{re}}\left(s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)$ as follows:

$$\begin{aligned}o^{\text{opt}}_{\text{re}}\left(s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right) &= \frac{f_{h+1}}{2} \cdot \left(\alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)^2 + \frac{1}{2} y^{\text{opt}}_{\min}\\ &= \frac{1}{2}\left(\sum_{\ell=1}^{h+1} \frac{f_\ell}{N_\ell\left(s^{(h+1)}_{i_{k-1},\ldots,i_h}\right)}\right) \left(\alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)^2.\end{aligned}$$

This shows that Equation (3.25) also holds for segment $s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}$ at level $h+1$ and, hence, completes the proof of the proposition. □

We are now ready to characterize the optimal k-level pattern. The result is stated in the following theorem.

Theorem 10. *A first-order approximation to the optimal k-level pattern and its overhead are characterized by*

$$W^{opt} = \sqrt{\frac{2\left(\sum_{\ell=1}^{k-1} N_\ell^{opt} C_\ell + C_k\right)}{\sum_{\ell=1}^{k-1} \frac{\lambda_\ell}{N_\ell^{opt}} + \lambda_k}}, \tag{3.29}$$

$$N_\ell^{opt} = \sqrt{\frac{\lambda_\ell}{C_\ell} \cdot \frac{C_k}{\lambda_k}}, \quad \forall \ell = 1, 2, \ldots, k-1, \tag{3.30}$$

$$H^{opt} = \sum_{\ell=1}^{k} \sqrt{2\lambda_\ell C_\ell} + O(\Lambda). \tag{3.31}$$

Proof. From Observation 3, Equation (3.22) and Proposition 9, we know that the optimal pattern can be obtained by minimizing the following function:

$$F = o_{\text{ef}} \cdot o_{\text{re}}^{\text{opt}} = \frac{1}{2}\left(\sum_{\ell=1}^{k-1} N_\ell C_\ell + C_k\right)\left(\sum_{\ell=1}^{k-1} \frac{f_\ell}{N_\ell} + f_k\right).$$

We first compute the optimal number of checkpoints at each level using a *two-phase iterative* method. Towards this end, let us define

$$o_{\text{ef}}(h) = \sum_{\ell=h}^{k-1} N_\ell C_\ell + C_k,$$
$$o_{\text{re}}^{\text{opt}}(h) = \frac{1}{2}\left(\sum_{\ell=h}^{k-1} \frac{f_\ell}{N_\ell} + f_k\right).$$

In the first phase, we set initially

$$F(1) = o_{\text{ef}}(1) \cdot o_{\text{re}}^{\text{opt}}(1).$$

The optimal value of N_1 that minimizes $F(1)$ can then be obtained by setting

$$\begin{aligned}\frac{\partial F(1)}{\partial N_1} &= C_1 o_{\text{re}}^{\text{opt}}(1) - o_{\text{ef}}(1)\frac{f_1}{2N_1^2} \\ &= C_1\left(\frac{f_1}{2N_1} + o_{\text{re}}^{\text{opt}}(2)\right) - \left(N_1 C_1 + o_{\text{ef}}(2)\right)\frac{f_1}{2N_1^2} \\ &= C_1 o_{\text{re}}^{\text{opt}}(2) - o_{\text{ef}}(2)\frac{f_1}{2N_1^2} = 0,\end{aligned}$$

which gives $N_1^{\text{opt}} = \sqrt{\frac{f_1}{C_1} \cdot \frac{o_{\text{ef}}(2)}{2o_{\text{re}}^{\text{opt}}(2)}}$. Substituting it into $F(1)$ and simplifying, we can get the value of F after the first iteration as

$$F(2) = \frac{1}{2}\left(\sqrt{f_1 C_1} + \sqrt{o_{\text{ef}}(2) \cdot o_{\text{re}}^{\text{opt}}(2)}\right)^2.$$

Repeating the above process, we can get the optimal value of F after $k-1$ iterations as

$$F^{\text{opt}} = F(k) = \frac{1}{2}\left(\sum_{\ell=1}^{k}\sqrt{f_\ell C_\ell}\right)^2, \tag{3.32}$$

and the optimal value of N_ℓ as

$$N_\ell^{\text{opt}} = \sqrt{\frac{f_\ell}{C_\ell}\cdot\frac{o_{\text{ef}}(\ell+1)}{2o_{\text{re}}^{\text{opt}}(\ell+1)}}, \quad \forall \ell = 1, 2, \ldots, k-1. \tag{3.33}$$

In the second phase, we first compute from Equation (3.33)

$$\begin{aligned} N_{k-1}^{\text{opt}} &= \sqrt{\frac{f_{k-1}}{C_{k-1}}\cdot\frac{o_{\text{ef}}(k)}{2o_{\text{re}}^{\text{opt}}(k)}} \\ &= \sqrt{\frac{f_{k-1}}{C_{k-1}}\cdot\frac{C_k}{f_k}} \\ &= \sqrt{\frac{\lambda_{k-1}}{C_{k-1}}\cdot\frac{C_k}{\lambda_k}}. \end{aligned}$$

Substituting it into N_{k-2}^{opt}, we obtain:

$$\begin{aligned} N_{k-2}^{\text{opt}} &= \sqrt{\frac{f_{k-2}}{C_{k-2}}\cdot\frac{o_{\text{ef}}(k-1)}{2o_{\text{re}}^{\text{opt}}(k-1)}} \\ &= \sqrt{\frac{\lambda_{k-2}}{C_{k-2}}\cdot\frac{N_{k-1}^{\text{opt}}C_{k-1}+C_k}{\frac{\lambda_{k-1}}{N_{k-1}^{\text{opt}}}+\lambda_k}} \\ &= \sqrt{\frac{\lambda_{k-2}}{C_{k-2}}\cdot\frac{\sqrt{\frac{\lambda_{k-1}}{\lambda_k}C_{k-1}C_k}+C_k}{\sqrt{\lambda_{k-1}\lambda_k\frac{C_{k-1}}{C_k}}+\lambda_k}} \\ &= \sqrt{\frac{\lambda_{k-2}}{C_{k-2}}\cdot\frac{C_k\left(\sqrt{\frac{\lambda_{k-1}}{\lambda_k}\cdot\frac{C_{k-1}}{C_k}}+1\right)}{\lambda_k\left(\sqrt{\frac{\lambda_{k-1}}{\lambda_k}\cdot\frac{C_{k-1}}{C_k}}+1\right)}} \\ &= \sqrt{\frac{\lambda_{k-2}}{C_{k-2}}\cdot\frac{C_k}{\lambda_k}}. \end{aligned}$$

Repeating the above process iteratively, we can compute the optimal values of N_ℓ^{opt} for $\ell = k-3, \ldots, 2, 1$, as given in Equation (3.30) by using values of $N_{k-1}^{\text{opt}}, \ldots, N_{\ell+1}^{\text{opt}}$.

The optimal pattern length, according to Equation (3.18), can be expressed as $W^{\text{opt}} = \sqrt{\frac{o_{\text{ef}}}{\Lambda\cdot o_{\text{re}}^{\text{opt}}}}$, which turns out to be Equation (3.29) with the optimal values of N_ℓ^{opt}.

The optimal overhead, according to Equations (3.19) and (3.32), can be expressed as $H^{\text{opt}} = 2\sqrt{\Lambda \cdot F^{\text{opt}}} + O(\Lambda)$, which gives rise to Equation (3.31). This completes the proof of the theorem. □

Since Proposition 9 shows that all the checkpoints of each level are equally spaced in the pattern, we can readily obtain the following corollary.

Corollary 2. *In an optimal k-level pattern, the number of level-ℓ checkpoints between any two consecutive level-$(\ell+1)$ checkpoints is given by*

$$n_\ell^{opt} = \frac{N_\ell^{opt}}{N_{\ell+1}^{opt}} = \sqrt{\frac{\lambda_\ell}{\lambda_{\ell+1}} \cdot \frac{C_{\ell+1}}{C_\ell}}. \tag{3.34}$$

for all $\ell = 1, \ldots, k-1$.

Remarks. The optimal k-level pattern derived in this section has a *rational* number of segments, while the optimal *integer* solution could be much harder to compute. In Section 3.4, we use rounding to derive a practical solution. Still, Equation (3.31) provides a lower bound on the optimal overhead, which is met very closely in all our experimental scenarios.

3.3.4 Optimal subset of levels

The preceding section characterizes the optimal pattern by using k levels of checkpoints. In many practical cases, there is no obligation to use all available levels. This section addresses the problem of selecting the optimal subset of levels in order to minimize the overall execution overhead.

Checkpoint cost models

So far, we have assumed that all the checkpoint costs are fixed under a multi-level checkpointing scheme. In practice, the checkpoint costs may vary depending upon the implementation, and upon the subset of selected levels. In order to determine the optimal subset, we identify the following two checkpoint cost models:

- **Fixed independent costs.** The checkpoint cost C_ℓ at level ℓ is the cost paid to save data at level ℓ, *independently* of the subset of levels used. In this model, the checkpoint costs stay the same for all possible subsets.
- **Incremental costs.** The checkpointing cost C_ℓ at level ℓ is the *additional* cost paid to save data when going from level $\ell - 1$ to ℓ. In this model, the checkpoint cost at a particular level depends on the subset of levels selected.

For example, with $k = 2$ levels and $C_1 = 10, C_2 = 20$, two subsets are possible: $\{1, 2\}$ and $\{2\}$. In the fixed independent cost model, these costs will stay unchanged regardless of the subset chosen. In the incremental cost model, since C_2 is the *additional* cost paid after C_1 is done, when using subset $\{2\}$, i.e., only placing level-2 checkpoints in the pattern, we need to

adjust its cost as $C_2' = 10 + 20 = 30$. In both cases, once the subset is decided, the checkpoint costs at the selected levels can be computed and therefore considered as fixed constants. The theoretical analysis presented in Section 3.3.3 can then be used to compute the optimal pattern.

But how to determine the optimal subset of levels? Consider again the example with $k = 2$ levels. In the incremental cost model, Equation (3.31) suggests that the optimal solution (ignoring lower-order terms) uses both levels if and only if

$$\begin{aligned} \sqrt{2\lambda_1 C_1} + \sqrt{2\lambda_2 C_2} &\le \sqrt{2\left(\lambda_1 + \lambda_2\right)\left(C_1 + C_2\right)} \\ \Leftrightarrow \quad 0 &\le \left(\sqrt{\lambda_1 C_2} - \sqrt{\lambda_2 C_1}\right)^2, \end{aligned}$$

which is always true when assuming $\lambda_1 \ge \lambda_2$ and $C_1 \le C_2$. We can easily apply the same argument to show that the optimal subset must contain *all* levels available as long as all checkpoint costs are positive.

In the fixed independent cost model, however, it is not clear whether all available levels should be used. Consider the same example with $k = 2$ levels, and define $\alpha = \frac{\lambda_2}{\lambda_1}$ and $\beta = \frac{C_2}{C_1}$. The optimal solution uses both levels if and only if

$$\begin{aligned} \sqrt{2\lambda_1 C_1} + \sqrt{2\lambda_2 C_2} &\le \sqrt{2\left(\lambda_1 + \lambda_2\right) C_2} \\ \Leftrightarrow \quad 4\alpha\beta &\le (\beta - 1)^2, \end{aligned}$$

which is not true when $\alpha = 0.5$ and $\beta = 2$. In this case, using only level-2 checkpoints leads to a smaller overhead.

Dynamic programming algorithm

In the fixed independent cost model, the optimal subset of levels in a general k-level pattern could well depend on the checkpoint costs and error rates of different levels. One can enumerate all 2^{k-1} possible subsets and select the one that leads to the smallest overhead. The following theorem presents a more efficient dynamic programming algorithm when the number k of levels is large.

Theorem 11. *Suppose there are k levels of checkpoints available and their costs are fixed. Then, the optimal subset of levels to use can be obtained by dynamic programming in $O(k^2)$ time.*

Proof. Let $\mathcal{S}^{\text{opt}}(h) \subseteq \{0, 1, \ldots, h\}$ denote the optimal subset of levels used by a pattern that is capable of handling errors up to level h, and let $H^{\text{opt}}(h)$ denote the corresponding optimal overhead (ignoring lower-order terms) incurred by the pattern. Define $\mathcal{S}^{\text{opt}}(0) = \emptyset$ and $H^{\text{opt}}(0) = 0$. Recall that $\Lambda_{[x,y]} = \sum_{\ell=x}^{y} \lambda_\ell$. We can compute $H^{\text{opt}}(h)$ using the following dynamic programming formulation:

$$H^{\text{opt}}(h) = \min_{0 \le \ell \le h-1} \left\{ H^{\text{opt}}(\ell) + \sqrt{2\Lambda_{[\ell+1,h]} C_h} \right\}, \tag{3.35}$$

and the optimal subset is $\mathcal{S}^{\text{opt}}(h) = \mathcal{S}^{\text{opt}}(\ell^{\text{opt}}) \bigcup \{h\}$, where ℓ^{opt} is the value of ℓ that yields the minimum $H^{\text{opt}}(h)$.

The optimal subset of levels to handle all k levels of errors is then given by $\mathcal{S}^{\text{opt}}(k)$ with the optimal overhead $H^{\text{opt}}(k)$. The complexity is clearly quadratic in the total number of levels. □

3.4 Simulations

In this section, we conduct a set of simulations whose goal is threefold: (i) to verify the accuracy of the first-order approximation; (ii) to confirm the optimality of the subset of levels found by the dynamic programming algorithm; and (iii) to evaluate the performance of our approach and to compare it with other multi-level checkpointing algorithms. After introducing the simulation setup in Section 3.4.1, we proceed in two steps. First, in Section 3.4.2, we instantiate the model with realistic parameters from the literature and run simulations for all possible subsets of levels and roundings. Then, in Section 3.4.3, we instantiate the model with different test cases from the recent work of Di et al. [37, 39] on multilevel checkpointing and compare the overheads obtained with three approaches: (a) Young/Daly's classical formula; (b) our first-order approximation formula; and (c) Di et al.'s iterative/optimal algorithm. The simulator code is publicly available at `http://perso.ens-lyon.fr/aurelien.cavelan/multilevel.zip`, so that interested readers can experiment with it and instantiate the model with parameters of their own choice.

3.4.1 Simulation setup

Checkpoint and recovery costs both depend on the volume of data to be saved, and are mostly determined by the hardware resource used at each level. As such, we assume that recovery cost for a given level is equivalent to the corresponding checkpointing cost, i.e., $R_\ell = C_\ell$ for $1 \leq \ell \leq k$ (unless specified otherwise). This a common assumption [37, 74], even though in practice the recovery cost can be expected to be *somewhat* smaller than the checkpoint cost [37, 39]. All costs are fixed and independent (as discussed in Section 3.3.4).

The simulator is fed with k levels of errors and their MTBFs $\mu_\ell = 1/\lambda_\ell$, as well as the resilience parameters C_ℓ and R_ℓ. For each of the 2^{k-1} possible subsets of levels (the last level is always included), we take the optimal pattern given in Theorem 10 and Corollary 2, and then try all possible roundings (floor and ceiling) based on the optimal (rational) number of checkpoints (n_ℓ^{opt} given in Equation (3.34)). For each rounding, we compare the following three overheads:

- **Simulated overhead**, obtained by running the simulation 10000 times and averaging the results;
- **Corresponding theoretical overhead**, obtained from Equations (3.19), (3.22) and (3.24) using the integer solution that corresponds to the rounding;
- **Theoretical lower bound**, obtained from Equation (3.31) with the optimal rational solution.

In the following, we associate Young/Daly's classical formula, defined as $W^{opt} = \sqrt{\frac{2C}{\Lambda}}$, with the highest checkpointing level available, i.e., $C = C_k$. Note that in this case, Young/Daly's formula and Equation (3.29) can be used interchangeably, and the corresponding theoretical overhead is obtained with $H^{opt} = \sqrt{2\Lambda C}$.

Table I
SETS OF PARAMETERS (A) AND (B) USED AS INPUTS FOR SIMULATIONS.

Set	From	Level	1	2	3	4
(A)	Moody et al. [74]	C (s)	0.5	4.5	1051	-
		MTBF (s)	5.00e6	5.56e5	2.50e6	-
(B)	Balaprakash et al. [6]	C (s)	10	30	50	150
		MTBF (s)	3.60e4	7.20e4	1.44e5	7.20e5

3.4.2 Assessing accuracy of first-order approximation

In this section, we run simulations with two sets of parameters, described in Table I. For each set of parameters, we consider all possible subsets of levels. Then, for each subset, we compute the optimal pattern length and number of checkpoints to be used at each level. We show the accuracy of our approach in both scenarios, and we confirm the optimality of the subset of levels returned by the dynamic programming algorithm.

Using set of parameters (A)

The first set of parameters (shown in set (A) of Table I) corresponds to the Coastal platform, a medium-sized HPC system of 1104 nodes at the Lawrence Livermore National Laboratory (LLNL). The Coastal platform has been used to evaluate the Scalable Checkpoint/Restart (SCR) library by Moody et al. [74], who provided accurate measurements for the checkpoint costs using real applications (given in the first row of Table I). There are $k = 3$ levels of checkpoints. First-level checkpoints are written to the local RAMs of the nodes, and this is the fastest method ($0.5s$). Second-level checkpoints are also written to local RAMs, but small sets of nodes collectively compute and store parity redundancy data, which takes a little while longer ($4.5s$). Lastly, Lustre is used to store third-level checkpoints onto the parallel file system, which takes significantly longer time ($1051s$). Failures were analyzed in [74], and the error rates are given in the second row of Table I. Note that the error rate at level 2 is higher than those of levels 1 and 3.

Results: Table II and Figure 3.4 present the simulation results. Table II shows, from left to right, the subset of levels used, the number of checkpoints computed by our first-order approximation formula for each possible rounding (N_1, N_2, N_3), the corresponding optimal pattern length ($W^{\text{opt}}(s)$), the simulated overhead (Sim. Ov.), the corresponding theoretical overhead (Th. Ov.), the absolute and relative differences of these two overheads (Ab. Diff. = 100 $\times$ (Sim. Ov. - Th. Ov.), and Rel. Diff. = 100 $\times$ (Sim. Ov. - Th. Ov.)/Sim. Ov.), and finally the theoretical lower bound for this subset (Th. L.B.).

With $k = 3$, there are four possible subsets of levels, and both the best simulated overhead and the corresponding theoretical overhead are achieved for the subset $\{2, 3\}$, with $N_2 = 35$ and $N_3 = 1$ (highlighted in **bold** in the table). First, the difference between the simulated and theoretical overheads is very small, with a difference $< 0.7\%$ in absolute values, and a relative difference ranging from 2.9% (for subset $\{1, 2, 3\}$) to 8.14% (for subset $\{3\}$), which shows the accuracy of the first-order approximation for this set of parameters. The simulated overhead is always higher than the theoretical one, which is expected, because the first-order approximation ignores some lower-order terms. Next, we observe that, for each subset, all roundings of the

Table II
SIMULATION RESULTS USING SET OF PARAMETERS (A).

Levels	N_1	N_2	N_3	W^{opt} (s)	Sim. Ov.	Th. Ov.	Abs. Diff.	Rel. Diff.	Th. L.B.
{3}	-	-	1	2.96e4	7.74e-2	7.11e-2	0.63%	8.14%	7.11e-2
{1,3}	14	-	1	3.09e4	7.40e-2	6.85e-2	0.55%	7.43%	6.85e-2
	13	-	1	3.09e4	7.39e-2	6.85e-2	0.54%	7.31%	
{2,3}	-	35	1	7.27e4	**3.44e-2**	**3.33e-2**	0.11%	3.20%	**3.33e-2**
	-	34	1	7.25e4	3.46e-2	**3.33e-2**	0.13%	3.76%	
{1,2,3}	33	33	1	7.27e4	3.46e-2	3.35e-2	0.11%	3.18%	3.35e-2
	32	32	1	7.24e4	3.45e-2	3.35e-2	0.10%	2.90%	

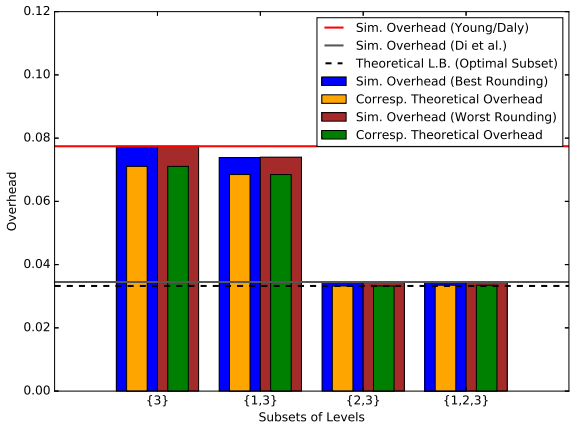


Figure 3.4: Simulated and (corresponding) theoretical overheads for all possible subsets of levels with the best and worst roundings for each subset using set of parameters (A).

number of checkpoints yield similar overheads on this platform, and the difference between the best and worst roundings is almost negligible.

Furthermore, using the best subset ($\{2, 3\}$) improves the overhead by over 50% compared to using level-3 checkpoints alone (as in Young/Daly's result). This is indeed the subset returned by the dynamic programming algorithm, and the result matches closely the minimum theoretical lower bound. Finally, comparing our result to the one obtained by the optimal two-level algorithm by Di et al. [39] on this best subset, we see that the simulated overheads are similar under the optimal subset, as the patterns found using both approaches share the same number of checkpoints and the pattern lengths are also almost identical.

Using set of parameters (B)

The second set of parameters correspond to the execution of the LAMMPS application on the large BG/Q platform Mira at the Argonne National Laboratory (ANL) [6]. The parameters are presented in set (B) of Table I. In this setting, the Fault Tolerance Interface (FTI) [12] was used, which has four checkpoint levels ($k = 4$): Local checkpoint; Local checkpoint + Partner-copy; Local checkpoint + Reed-Solomon coding; and PFS-based checkpoint. The MTBFs

Table III
SIMULATION RESULTS USING SET OF PARAMETERS (B).

Levels	N_1	N_2	N_3	N_4	W^{opt} (s)	Sim. Ov.	Th. Ov.	Abs. Diff.	Rel. Diff.	Th. L.B.
{4}	-	-	-	1	2.45e3	1.43e-1	1.22e-1	1.9%	13.3%	1.22e-1
{1,4}	5	-	-	1	3.79e3	1.18e-1	1.05e-1	1.3%	11.0%	1.05e-1
	4	-	-	1	3.61e3	1.18e-1	1.05e-1	1.3%	11.0%	
{2,4}	-	5	-	1	6.00e3	1.11e-1	1.00e-1	1.1%	9.9%	1.00e-1
{3,4}	-	-	11	1	1.55e4	9.96e-2	9.02e-2	0.94%	9.44%	9.01e-2
	-	-	10	1	1.44e4	9.91e-2	9.01e-2	0.90%	9.08%	
{1,2,4}	9	3	-	1	6.41e3	1.11e-1	1.03e-1	0.8%	7.2%	1.02e-1
	6	2	-	1	5.21e3	1.13e-1	1.04e-1	0.9%	8.0%	
	6	3	-	1	5.84e3	1.11e-1	1.03e-1	0.8%	7.2%	
	4	2	-	1	4.74e3	1.17e-1	1.05e-1	1.2%	10.3%	
{1,3,4}	21	-	7	1	1.58e4	9.72e-2	8.99e-2	0.73%	7.51%	**8.96e-2**
	18	-	6	1	1.40e4	9.82e-2	**8.98e-2**	0.84%	8.55%	
	14	-	7	1	1.04e4	**9.68e-2**	9.01e-2	0.67%	6.92%	
	12	-	6	1	1.26e4	9.85e-2	9.04e-2	0.81%	8.22%	
{2,3,4}	-	16	4	1	1.70e4	1.07e-1	9.75e-2	0.95%	8.9%	9.68e-2
	-	12	3	1	1.36e4	1.04e-1	9.73e-2	0.67%	6.4%	
	-	12	4	1	1.47e4	1.05e-1	9.68e-2	0.82%	7.8%	
	-	9	3	1	1.17e4	1.05e-1	9.75e-2	0.75%	7.1%	
{1,2,3,4}	24	8	4	1	1.66e4	1.09e-1	1.00e-1	0.9%	8.2%	9.92e-2
	18	6	3	1	1.32e4	1.08e-1	9.99e-2	0.81%	7.5%	
	12	4	4	1	1.15e4	1.11e-1	1.03e-1	0.8%	7.2%	
	9	3	3	1	9.17e3	1.14e-1	1.05e-1	0.9%	7.9%	
	16	8	4	1	1.51e4	1.08e-1	9.95e-2	0.85%	7.9%	
	12	6	3	1	1.20e4	1.09e-1	1.00e-1	0.9%	8.3%	
	8	4	4	1	1.05e4	1.16e-1	1.05e-1	1.1%	9.5%	
	6	3	3	1	8.33e3	1.19e-1	1.08e-1	1.1%	9.2%	

correspond to the failure rates typically observed for petascale HPC applications [12, 37, 74].

Results: Table III and Figure 3.5 present the simulation results for this set of parameters. There are 8 possible subsets of levels. As before, we observe that the theoretical overhead is always slightly smaller than the simulated one, with an absolute difference of less than 2%, and a relative difference between 6-14$\%$, demonstrating the accuracy of the model. Again, the results are very close to the theoretical lower bound. For this platform, the simulated overheads vary from 9.68% (with optimal subset of levels $\{1, 3, 4\}$ found by the dynamic programming algorithm) to 14.3% (with level-4 checkpoints alone). For a given subset of levels, the rounding does not play a significant role, as W^{opt} is also adjusted accordingly (increased or decreased) as a result of rounding. For instance, we observe that, for subset $\{1, 2, 3, 4\}$, the numbers of checkpoints at levels 1 and 2 are halved for the third rounding compared to the first rounding in Table III, but W^{opt} is also reduced by 31%, so that for the same amount of work, the number of checkpoints does not change by much. We can also see that the pattern length W^{opt} for the

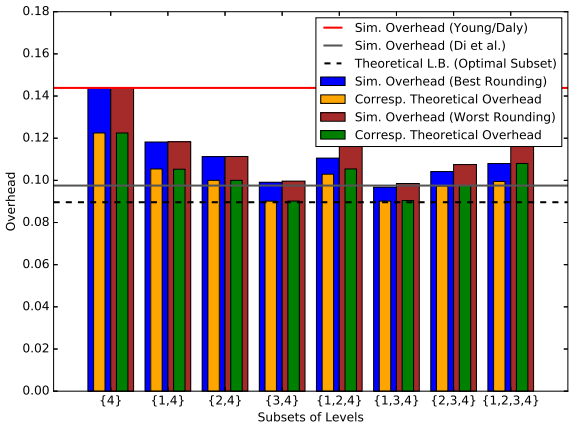


Figure 3.5: Simulated and (corresponding) theoretical overheads for all possible subsets of levels with the best and worst roundings for each subset using set of parameters (B).

smallest overhead is around $10400s$, but only $2450s$ for the largest overhead. In fact, the largest pattern lengths are obtained for the *highest cumulated checkpoint cost*, which turns out to be $830s$ for $\{2,3,4\}$ with $N_2=16, N_3=4, N_4=1$, and for $\{1,2,3,4\}$ with $N_1=24, N_2=8, N_3=4$ and $N_4=1$. This is because using more checkpoints both increases the error-free overhead and reduces the time lost due to re-executions upon errors. As a consequence, and to mitigate the aforementioned overhead, the length of the pattern increases (e.g., $W^{\text{opt}} = 17000s$ for $\{2,3,4\}$ and $W^{\text{opt}} = 16600s$ for $\{1,2,3,4\}$). And the converse is also true: when using fewer checkpoints, the error-free overhead decreases and the time lost upon errors increases. In order to compensate, the pattern length decreases (e.g., $W^{\text{opt}} = 8330s$ for $\{1,2,3,4\}$ with $N_1=6, N_2=3, N_3=3$ and $N_4=1$).

We note that, in this case, our first-order solution slightly outperforms the iterative method by Di et al. [37] on multi-level checkpointing (with a simulated overhead of 9.68e-2 compared to 9.75e-2). The reason is that their algorithm computes a solution under the independent checkpointing model, i.e., checkpoints at different levels are taken according to different independent periods. However, it is not clear how such a model can be implemented in practice due to the difficulties as explained in Section 3.1 and the different options to rollback to a checkpoint in case of a fault. Therefore, we transformed their result to a pattern-based solution by rounding the different numbers of checkpoints obtained using their algorithm to create equal number of checkpoints at level $\ell-1$ between two consecutive level-ℓ checkpoints. Although the best rounding is selected here for comparison, the result can still change drastically the number of checkpoints computed by their initial rational solution without changing the pattern length, thus increasing the overhead.

3.4.3 Comparing performance of different approaches

In this section, we conduct simulations using settings from Di et al.'s recent work on multi-level checkpointing, which comprises two cases with four levels [37] and eight cases with two levels [39], thus covering a wide range of configurations. For each case, we compare the per-

formance of three different approaches: (a) Young/Daly's classical formula; (b) our first-order approximation formula; and (c) Di et al.'s iterative algorithm.

Table IV
SET OF PARAMETERS (C) USED AS INPUT FOR SIMULATIONS.

Set (C), from Di et al. [37]					
	Level	1	2	3	4
Case #A	$C\ (s)$	8	10	80	90
	$R\ (s)$	8	10	80	90
	MTBF (s)	2160	1440	8640	21600
Case #B	$C\ (s)$	1	20	60	70
	$R\ (s)$	1	10	30	35
	MTBF (s)	864	864	1080	1440

Using set of parameters (C)

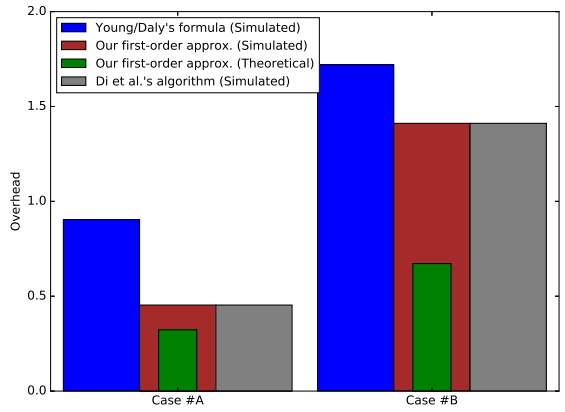


Figure 3.6: Performance comparison of the three different approaches using two cases from Di et al. [37].

We first run simulations for Cases #A and #B, whose parameters are presented in Table IV. These parameters are based on the FTI multilevel checkpointing model and have been used by Di et al. [37] to evaluate the performance of their approach. Note that the recovery cost is about half that of the checkpointing cost in Case #B.

In their work, Di et al. considered independent checkpointing periods, as opposed to the nested method based on periodic patterns (as discussed in Section 3.1). Although they provided an optimal solution, an iterative approach was used to compute it numerically in contrast to the simple formula we propose in this work. Recall that using independent checkpointing periods allows checkpoints at different levels to be taken simultaneously, which can hardly be done in practice. Adapting their solution to our model results in rational numbers of checkpoints, and we again use rounding to resolve this issue. We find that, using the best roundings for

both approaches, their solution turns out to be very similar to ours (with the same number of checkpoints, and close periods with $< 1\%$ difference).

Results: Figure 3.6 presents the overheads for both cases. First, we observe that Di et al.'s optimal iterative algorithm has almost identical performance to our solution, with a simulated overhead around 45% for Case #A and 140% for Case #B under both approaches. However, using Young/Daly's formula to checkpoint only at the highest level yields significantly worse overheads (around 90% for Case #A and 170% for Case #B). Overall, our solution is as good as Di et al.'s optimal numerical one (but has much less complexity), and it is up to 45% better than Young/Daly's formula in Case #A and 30% better in Case #B.

Note that the corresponding theoretical overhead of our solution is close to the simulated one for Case #A, but starts to diverge for Case #B. This is because first-order approximation is only accurate when the resilience parameters and pattern length are small compared to the MTBF, which is no longer true for Case #B. Specifically, we have:

- In Case #A, the optimal subset of levels is $\{2, 4\}$. The optimal pattern has length $W^{\text{opt}} = 1052s$ and consists of $N_2 = 8$ level-2 checkpoints followed by $N_4 = 1$ level-4 checkpoint, meaning that we have a level-2 checkpoint every $131.5s$ of computation. So a level-2 checkpoint is saved every $141.5s$ and a level-4 checkpoint is saved every 1222s. On the other hand, the combined MTBF for errors at levels 1 and 2 (handled by level-2 checkpoints) is $864s$ and the combined MTBF for errors at levels 3 and 4 (handled by level-4 checkpoints) is $6171s$. Hence, we have $\frac{141.5}{864} = 0.164$ and $\frac{1222}{6171} = 0.198$, which are reasonably small, making our solution accurate.

- In Case #B, the optimal subset of levels is $\{1, 4\}$, and the optimal pattern has $W^{\text{opt}} = 223s$, $N_1 = 5$ and $N_4 = 1$. Thus, we have a level-1 checkpoint every $44.6s$ of computation. So a level-1 checkpoint is saved every $45.6s$ and a level-4 checkpoint is saved every $298s$. The MTBF for errors at level 1 is $864s$ and the combined MTBF for errors at levels 2, 3 and 4 (handled by level-4 checkpoints) is $360s$. Thus, we have $\frac{44.6}{864} = 0.052$, which is fine, but $\frac{298}{360} = 0.828$, which is too high and essentially makes the first-order solution inaccurate.

Despite the difference between the theoretical and simulated overheads under Case #B, the proximity of our solution to Di et al.'s optimal numerical solution nevertheless shows the usefulness of first-order approximation for determining the optimal multi-level checkpointing patterns.

Using set of parameters (D)

Finally, we run simulations for eight cases, whose parameters are presented in Table V. These parameters have been used by Di et al. [39] to evaluate their two-level checkpointing model, and as such, each case consists of only two checkpointing levels. In their work, the authors proposed an optimal solution by solving complex mathematical equations using numerical method. Again, for each case, we compare the simulated overheads obtained with the three different approaches.

Table V
SET OF PARAMETERS (D) USED AS INPUT FOR SIMULATIONS.

Set (D), from Di et al. [39]							
	Level	1	2		Level	1	2
Case 1	$C\ (s)$	20	50	Case 5	$C\ (s)$	10	40
	MTBF (s)	3600	21600		MTBF (s)	432	2160
Case 2	$C\ (s)$	20	50	Case 6	$C\ (s)$	100	20
	MTBF (s)	1728	8640		MTBF (s)	432	2160
Case 3	$C\ (s)$	20	100	Case 7	$C\ (s)$	40	200
	MTBF (s)	864	4320		MTBF (s)	288	1440
Case 4	$C\ (s)$	10	40	Case 8	$C\ (s)$	50	300
	MTBF (s)	864	4320		MTBF (s)	216	1440

In this set of parameters, the MTBF has a large variation, ranging from more than 1 hour (Case 1) to less than 4 minutes (Case 8). Similarly, the checkpointing costs vary from $10s$ (Cases 4 and 5) to $300s$ (Case 8). Note that Cases 7 and 8 have both very short MTBFs and very high checkpointing costs, resulting in a lot of errors and recoveries. In particular, the checkpointing cost at level 2 in Case 8 ($300s$) is larger than the MTBF at level 1 ($216s$).

Results: Figure 3.7 presents the simulation results for the eight cases. First, we observe that the optimal algorithm by Di et al. only yields a slightly better simulated overhead compared to our simple first-order approximation solution (by less than 2% in Cases 1 to 6). However, our solution always improves significantly over Young/Daly's formula, from 2% (Case 1) up to 100% (Case 6). Due to their short MTBFs, Cases 7 and 8 stand out and incur much higher overheads compared to the first six cases (thus their results are presented in a separate plot). Still, considering Case 8, we are able to improve over Young/Daly's solution by as much as 2500% (in absolute value of the overhead), and we are off the optimal simulated overhead by only 300%. In addition, Figure 3.7 shows the theoretical overheads obtained both with our formula and the solution provided by Di et al. in [39]. As expected, our first-order approximation remains accurate when the MTBF is large, as in Cases $1, 2$ and 4. However, it becomes less accurate with shorter MTBFs and higher error rates, especially in Cases 7 and 8 (which do not represent healthy HPC platforms).

3.4.4 Summary of results

From the simulation results, we conclude that first-order approximation remains a valuable performance model for evaluating checkpointing solutions in HPC systems (as long as the error rates stay reasonably low). We have demonstrated, through an extensive set of simulations with a wide range of parameters, the usefulness of multi-level checkpointing (over using only one level of checkpoints) with significantly reduced overheads. The results also corroborate the analytical study by showing the benefit of selecting an optimal subset of levels among all the levels available. Overall, our approach achieves the optimal or near-optimal performance in almost all cases, except when the MTBF is too small, in which case even the optimal solution

yields an unacceptably high overhead (e.g., Case 8 of Table V).

3.5 Conclusion

This chapter has studied multi-level checkpointing protocols, where different levels of checkpoints can be set; lower levels deal with frequent errors that can be recovered at low cost (for instance with a memory copy), while higher levels allow us to recover from all errors, such as node failures (for instance with a copy in stable storage). We consider a general scenario with k levels of faults, and we provide explicit formulas to characterize the optimal checkpointing pattern, up to first-order approximation. The overhead turns out to be of the order of $\sum_{\ell=1}^{k} \sqrt{2\lambda_\ell C_\ell}$, which elegantly extends Young/Daly's classical formula.

The first-order approximation to the optimal k-level checkpointing pattern uses rational numbers of checkpoints, and we prove that all segments should have equal lengths. We corroborate the theoretical study by an extensive set of simulations, demonstrating that greedily rounding the rational values leads to an overhead very close to the lower bound. Furthermore, we provide a dynamic programming algorithm to determine those levels that should be selected, and the simulations confirm the optimality of the subset of levels returned by the dynamic programming algorithm.

The problem of finding a first-order optimal pattern with an integer number of segments to minimize the overhead remains open. It may well be the case that such an integer pattern is not periodic at each level and uses different-length segments. However, the good news is that the rounding of the rational solution provided in this chapter seems quite efficient in practice.

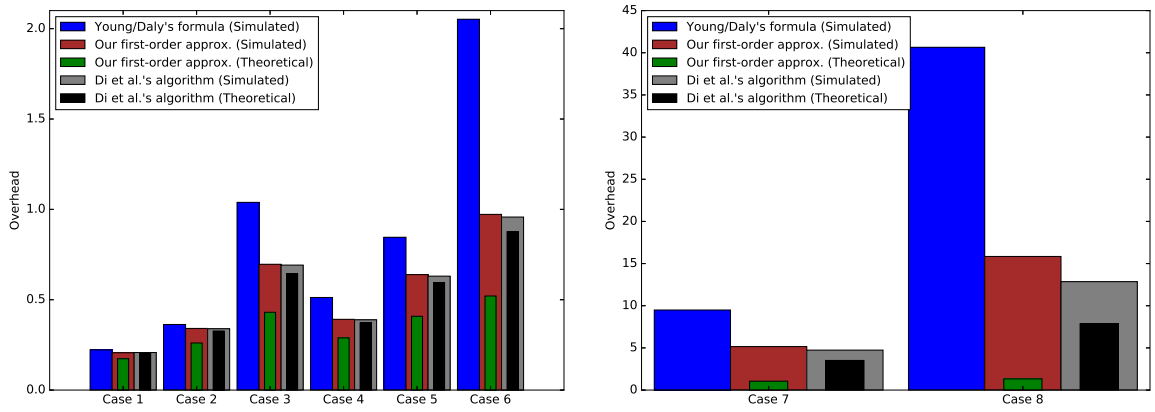


Figure 3.7: Performance comparison of the three different approaches using 8 cases from Di et al. [39].

Part II

Application Workflows

Chapter 4

Multi-level Checkpointing and Verification for Linear Workflows

This chapter focuses on High Performance Computing (HPC) workflows whose dependency graph forms a linear chain. This work extends and generalizes a preliminary analysis [J3], as well as a more recent work [W5]. Similarly to Chapters 2 and 3, we extend single-level checkpointing in two important directions. Our first contribution targets silent errors, and combines in-memory checkpoints with both partial and guaranteed verifications. Our second contribution deals with multi-level checkpointing for fail-stop errors. We present sophisticated dynamic programming algorithms that return the optimal solution for each problem in polynomial time. We also show how to combine all these techniques and solve the general problem with both fail-stop and silent errors. Simulation results demonstrate that these extensions lead to significantly improved performance compared to the standard single-level checkpointing algorithm. The work in this chapter has been published in *Journal of computational science* (JoCS) [J4].

4.1 Introduction

Multilevel checkpointing is now the state-of-the-art technique when it comes to dealing with fail-stop errors. In Part I, we have introduced both partial verifications, two-level checkpointing for silent errors and fail-stop errors, and a generalization of multi-level checkpointing for fail-stop errors only. In this chapter, we follow the same approach, and we use a similar model as the one presented in Chapter 3, but we consider linear workflow applications rather than divisible applications.

We first consider a very general scenario, where the platform is subject to k levels of fail-stop errors, numbered from 1 to k. Level ℓ is associated with an error rate λ_ℓ, a checkpointing cost $C^{(\ell)}$, and a recovery cost $R^{(\ell)}$. A fault at level ℓ destroys all the checkpoints of lower levels (from 1 to $\ell - 1$ included) and implies a rollback to a checkpoint of level ℓ or higher. Similarly, a recovery of level ℓ will restore data from all lower levels. As mentioned, fault rates are decreasing and checkpoint/recovery costs are increasing when we go to higher levels: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k$, $C^{(1)} \leq C^{(2)} \leq \cdots \leq C^{(k)}$, and $R^{(1)} \leq R^{(2)} \leq \cdots \leq R^{(k)}$. The problem is to determine the optimal locations to place checkpoints of various levels in a High-

Performance Computing (HPC) application.

Regarding silent errors, a traditional checkpointing strategy can still be used, provided that it is coupled with a verification mechanism to detect silent errors (either partial or guaranteed), as seen Chapter 1. Furthermore, rather than checkpointing on stable storage (e.g., an external disk), a lightweight mechanism of in-memory checkpoints can be provided to cope with silent errors: one keeps a local copy of the data that has not been corrupted when a silent error strikes, and it can be used to perform a recovery rapidly. However, such local copies are lost once a fail-stop error occurs, and hence checkpoints on stable storage must also be provided when dealing with both sources of errors.

Designing resilience algorithms by combining all of these techniques is quite challenging. In this chapter, we deal with a simplified, yet realistic, application framework, where a set of application workflows exchange data at the end of their execution. Such a framework can be modeled as a task graph whose dependencies form a linear chain. This scenario corresponds to an HPC application whose workflow is partitioned into a succession of (typically large) tightly-coupled computational kernels, each of which is identified as a task. Hence, we consider a linear chain of tasks $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$, where each task T_i ($1 \leq i \leq n$) has a weight w_i corresponding to its computational load. The following summarizes our approach to enforcing resilience in this simplified application framework:

Silent errors. To cope with silent errors, we couple in-memory checkpoints with both partial and guaranteed verifications. At the end of each task, we can perform either a partial verification (with cost V) or a guaranteed verification (with cost V^*) of the task output; or, probably less frequently, we can perform a guaranteed verification followed by a memory checkpoint (with cost C^M). Note that we do not take the risk of storing a corrupted checkpoint, hence the need for a guaranteed verification.

Fail-stop errors. To cope with fail-stop errors, we use general multi-level checkpointing and schedule checkpoints of various levels at the end of carefully selected tasks. Checkpoints of level 1 are inserted more frequently than checkpoints of level 2, which themselves are more frequent than checkpoints of level 3, and so on. In our approach, assuming that all checkpointing levels are used, a checkpoint at level ℓ is always preceded by checkpoints at all lower levels 1 to $\ell - 1$, which makes good sense in practice (e.g., with two levels, say local SSD and remote disk, one writes the data onto the local SSD before transferring it to remote the disk). In this context, the checkpointing cost $C^{(\ell)}$ at level ℓ is the cost paid to save data when going from level $\ell - 1$ to level ℓ.

Both error sources. To cope with both fail-stop and silent errors, we combine all these techniques: partial and guaranteed verifications, in-memory checkpointing, and several additional levels of checkpointing.

The main contributions of this chapter are several sophisticated dynamic programming algorithms that return the optimal solution for each of the three problems above, i.e., the solution that minimizes the expected execution time of the task chain in polynomial time. To the best of our knowledge, this is the first work that combines multi-level checkpointing with guaranteed and partial verifications to deal with both fail-stop and silent errors in linear workflows.

Furthermore, we present extensive simulations that demonstrate the usefulness of mixing these techniques, and, in particular, we demonstrate the gain obtained thanks to additional verifications and multi-level checkpointing. We show that it may be beneficial to use only some of the checkpointing levels; in this case they are renumbered from 1 to k. The best combination of levels to use can be found by an exhaustive search, since the number of levels k is usually small (3 or 4).

The rest of this chapter is organized as follows. We present the dynamic programming algorithm for silent errors with memory checkpoints and verifications in Section 4.2 and that for fail-stop errors in Section 4.3. The solution to deal with both error sources is described in Section 4.4. Simulation results are presented in Section 4.5. We survey related work in Section 4.6. Finally, we give concluding remarks and hints for future work in Section 4.7.

4.2 Memory checkpointing and verifications for silent errors

In this section, we present a sophisticated dynamic programming algorithm to decide which tasks to checkpoint and which tasks to verify. We first introduce the model in Section 4.2.1. We describe in Section 4.2.2 a dynamic programming algorithm for the case where only verified memory checkpoints are taken (i.e., memory checkpoints preceded by a guaranteed verification). We show how to extend this algorithm to add additional guaranteed verifications between checkpoints in Section 4.2.3, and finally we deal with the more complex case of partial verifications in Section 4.2.4.

4.2.1 Model

We consider a chain $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ of n tasks that execute on a large-scale platform subject to silent errors. Each task T_i is associated with a computational load or weight w_i, which is assumed to be known to the algorithm. For notational convenience, we define $W_{]i,j]} = \sum_{p=i+1}^{j} w_p$ to be the total weight of tasks T_{i+1} to T_j for any $0 \leq i < j \leq n$. The arrival times of silent errors follow a *Poisson process* with error rate λ_s. Unlike fail-stop errors, silent errors do not destroy the memory content when they strike. Hence, we can cope with silent errors by using lightweight memory checkpoints. When a silent error is detected, either by a partial verification or by a guaranteed one, we roll back to the nearest memory checkpoint, and recover from the memory copy there. This is much cheaper than checkpointing on and recovering from a disk checkpoint. We enforce that a guaranteed verification is always taken immediately before each memory checkpoint, so that all checkpoints are valid, and hence only one checkpoint needs to be maintained at any time during the execution of the application. Furthermore, we assume that the costs of checkpointing, recovery and verifications are uniform across different tasks, and that they are protected from faults (i.e., silent errors only strike the computations).

Let C^M denote the cost of memory checkpointing and R^M the cost of memory recovery. Also, let V^* denote the cost of guaranteed verification and V the cost of a partial verification. The accuracy of a partial verification is measured by its *recall*, which is denoted by r and

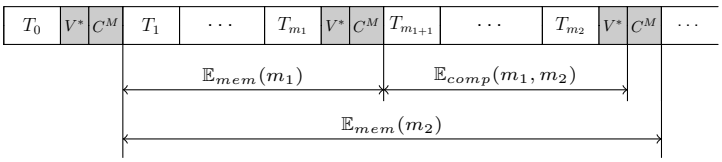


Figure 4.1: Placing verified memory checkpoints only: m_2 is fixed, and we try all possible locations for an additional verified memory checkpoint at m_1 between T_0 and T_{m_2}. Note that all subproblems $\mathbb{E}_{mem}(m_1)$, with $0 \leq m_1 < m_2$, have already been computed.

represents the proportion of detected errors over all silent errors that have occurred during the execution.[1] For notational convenience, we define $g = 1 - r$ to be the proportion of undetected errors. Note that the guaranteed verification can be considered as one with recall $r^* = 1$. Since a partial verification usually incurs a much smaller cost and yet has a reasonable recall [10, 15], it is highly attractive for detecting silent errors, and we make use of them between guaranteed verifications. For convenience, we introduce before task T_1 a *virtual* task T_0, which is checkpointed, and whose recovery cost is zero. This accounts for the fact that it is always possible to restart the application from scratch (i.e., recover from T_0) at no extra cost.

The SILENT problem consists in finding the optimal set of tasks to checkpoint as well as the optimal set of tasks to verify, along with the type of verification (guaranteed or partial) that should be applied. The objective is to minimize the total expected execution time of the task chain.

4.2.2 With memory checkpoints only

In this section, we present a dynamic programming algorithm when using only verified memory checkpoints (memory checkpoints preceded by a guaranteed verification). A naive brute-force algorithm would have to try all possible solutions (i.e., deciding whether or not to add a checkpoint after each task), resulting in 2^n operations. However, by remarking that the problem can be divided into independent subproblems, we can compute the optimal solution in polynomial time:

Theorem 12. *The optimal solution to the* SILENT *problem with only memory checkpoints can be obtained using a dynamic programming algorithm in $O(n^2)$ time and $O(n)$ space, where n is the number of tasks in the chain.*

The remainder of this section is devoted to proving this theorem. We first detail how the dynamic programming algorithm places the memory checkpoints in Section 4.2.2, and then we detail the computation of expected execution time between two memory checkpoints in Section 4.2.2. Finally, we provide the algorithm complexity in Section 4.2.2.

[1] Another measure of accuracy for a partial verification is *precision*, which is denoted as p and represents the proportion of true errors over all silent errors that are reported by the verification. Typically, a tradeoff exists between the recall and precision in the design of a partial verification mechanism. In this chapter, we assume perfect precision, i.e., $p = 1$, which has been shown to represent the most useful configuration for optimizing the execution overhead in Chapter 1.

Placing memory checkpoints

We define $\mathbb{E}_{mem}(m_2)$ as the optimal expected time needed to successfully execute all tasks from T_1 to T_{m_2}, where there is a verified memory checkpoint after task T_{m_2}. The goal is to obtain:

$$\mathbb{E}_{mem}(n) \ ,$$

which is the optimal expected execution time needed to successfully execute all the tasks in the chain. Intuitively, the idea is to compute $\mathbb{E}_{mem}(0), \mathbb{E}_{mem}(1), \mathbb{E}_{mem}(2), \ldots, \mathbb{E}_{mem}(n)$, in this order, so that we can compute each new value by reusing previously computed optimal results. We use memorization: we store and reuse solutions to subproblems instead of recomputing them. This is possible because $\mathbb{E}_{mem}(i)$, for $0 \leq i < n$, does not depend on tasks T_{i+1} to T_n. In other words, $\mathbb{E}_{mem}(i)$ can be used as an independent subproblem, which we compute once and then reuse to solve all $\mathbb{E}_{mem}(j)$ problems, with $i < j \leq n$.

To compute the general subproblem $\mathbb{E}_{mem}(m_2)$, we try all possible locations for an additional intermediate verified memory checkpoint between tasks T_0 and T_{m_2}, as illustrated in Figure 4.1. For each possible location m_1, we can reuse the optimal result given by $\mathbb{E}_{mem}(m_1)$, and we call $\mathbb{E}_{comp}(m_1, m_2)$, the expected time needed to successfully execute tasks T_{m_1+1} to T_{m_2}, knowing that there is no intermediate checkpoint. Finally, we add the cost of the checkpoint C^M following T_{m_2} (note that we account for the cost of the verification in $\mathbb{E}_{comp}(m_1, m_2)$), and we can write:

$$\mathbb{E}_{mem}(m_2) = \min_{0 \leq m_1 < m_2} \left\{ \mathbb{E}_{mem}(m_1) + \mathbb{E}_{comp}(m_1, m_2) \right\} + C^M \ ,$$

which is initialized with:

$$\mathbb{E}_{mem}(0) = 0 \ .$$

Indeed, when $m_2 = 0$, there is no task to execute, and no room for extra checkpoints.

Computing $\mathbb{E}_{comp}(m_1, m_2)$

Now, to compute the expected time needed to successfully execute several tasks between two verified memory checkpoints, we need only the position of the last verified memory checkpoint m_1, and the position of the next verified memory checkpoint m_2.

First, we pay the cost $W_{]m_1,m_2]}$ to execute all the tasks from T_{m_1+1} to T_{m_2}. Then, we pay the cost of the guaranteed verification V^*. There is a probability $p^s_{]m_1,m_2]} = 1 - e^{-\lambda_s W_{]m_1,m_2]}}$ of detecting a silent error, in which case we recover from the last verified memory checkpoint at m_1, with cost R^M (set to 0 if $m_1 = 0$) and we re-execute all the tasks from there, which is simply $\mathbb{E}_{comp}(m_1, m_2)$. Therefore, we have:

$$\mathbb{E}_{comp}(m_1, m_2) = W_{]m_1,m_2]} + V^* + p^s_{]m_1,m_2]} \left(R^M + \mathbb{E}_{comp}(m_1, m_2) \right) \ .$$

Simplifying the equation above and solving for $\mathbb{E}_{comp}(m_1, m_2)$, we obtain:

$$\mathbb{E}_{comp}(m_1, m_2) = e^{\lambda_s W_{]m_1,m_2]}} \left(W_{]m_1,m_2]} + V^* \right) + \left(e^{\lambda_s W_{]m_1,m_2]}} - 1 \right) R^M \ .$$

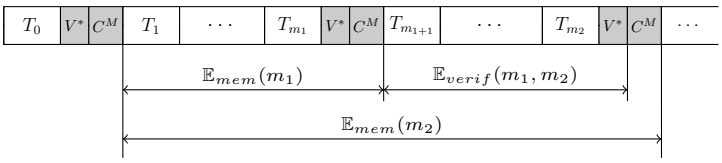


Figure 4.2: Placing memory checkpoints (with guaranteed verifications): m_2 is fixed, and we try all possible locations for m_1. Note that all subproblems $\mathbb{E}_{mem}(m_1)$, with $0 \leq m_1 < m_2$, have already been computed, while $\mathbb{E}_{verif}(m_1, m_2)$ is computed by yet another dynamic programming level to be described later (see Figure 4.3).

Complexity

The complexity is dominated by the computation of the table $\mathbb{E}_{mem}(m_1)$, which contains $O(n)$ entries, and each entry depends on at most n other entries that are already computed. Hence, the overall complexity of the algorithm is $O(n^2)$ in time and $O(n)$ in space. Note that each entry is computed only once using memoization, a well-known technique in dynamic programming [34] that leads to a recursive algorithm whose cost is the same as its iterative counterpart.

4.2.3 With memory checkpoints and guaranteed verifications

In this section, we extend the dynamic programming algorithm presented in Section 4.2.2 to allow for additional intermediate guaranteed verifications between two (verified) memory checkpoints:

Theorem 13. *The optimal solution to the* SILENT *problem with memory checkpoints and intermediate guaranteed verifications can be obtained using a dynamic programming algorithm in $O(n^3)$ time and $O(n^2)$ space, where n is the number of tasks in the chain.*

Figures 4.2 and 4.3 illustrate the idea of the algorithm, which now contains two dynamic programming levels, responsible for placing memory checkpoints (Figure 4.2) and guaranteed verifications (Figure 4.3), respectively. In the first level, $\mathbb{E}_{comp}(m_1, m_2)$ is replaced by $\mathbb{E}_{verif}(m_1, m_2)$, which accounts for additional guaranteed verifications. The remainder of this section is devoted to proving this theorem.

Placing memory checkpoints

The first level is the same as before (see Section 4.2.2), except that instead of calling $\mathbb{E}_{comp}(m_1, m_2)$ to compute the expected execution time between two memory checkpoints, we now call $\mathbb{E}_{verif}(m_1, m_2)$ to try and place additional guaranteed verifications in this interval (see Figure 4.2). We can write:

$$\mathbb{E}_{mem}(m_2) = \min_{0 \leq m_1 < m_2} \left\{ \mathbb{E}_{mem}(m_1) + \mathbb{E}_{verif}(m_1, m_2) \right\} + C^M \ ,$$

which is initialized by:

$$\mathbb{E}_{mem}(0) = 0 \ .$$

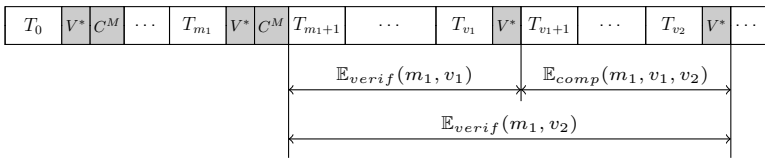


Figure 4.3: Placing guaranteed verifications: m_1 and v_2 are fixed, and we try all possible locations for an additional guaranteed verification at v_1 between T_{m_1} and T_{v_2}. Note that all subproblems $\mathbb{E}_{verif}(m_1, v_1)$, with $m_1 \leq v_1 < v_2$, have already been computed.

Placing guaranteed verifications

The second level searches where to insert additional guaranteed verifications between the last memory checkpoint at m_1 and the next guaranteed verification at v_2. We define $\mathbb{E}_{verif}(m_1, v_2)$ as the expected execution time needed to sucessfully execute tasks from T_{m_1+1} to T_{v_2}. The function is first called from the first level between two memory checkpoints with $v_2 = m_2$ (as $\mathbb{E}_{verif}(m_1, m_2)$), each of which also comes with a guaranteed verification. The approach to solve the problem is the same as before: suppose $m_1 = 0$, we compute all $\mathbb{E}_{verif}(0, 1), \mathbb{E}_{verif}(0, 2), \mathbb{E}_{verif}(0, 3)$, in this order, so that we can compute each step by reusing all previously computed optimal results as before. But because we have n possible values for m_1, we effectively need to solve this problem n times.

Again, this is possible because $\mathbb{E}_{verif}(m_1, v_2)$, $0 \leq m_1 < v_2 < n$ does not depend upon tasks T_{v_2+1} to T_n. In other words, $\mathbb{E}_{verif}(m_1, v_2)$ can be used as an independent subproblem, which is computed exactly once for each (m_1, v_2). These values are stored in a 2D table, so that we can reuse them to solve the larger problems as we progress from T_0 to T_n.

In order to solve $\mathbb{E}_{verif}(m_1, v_2)$, we try all possible locations for the last verification between T_{m_1} and T_{v_2}, and for each possible location v_1, we can reuse the optimal result given by $\mathbb{E}_{verif}(m_1, v_1)$, and we need to compute the expected time needed to successfully execute the tasks between two guaranteed verifications from T_{v_1+1} to T_{v_2}, denoted by $\mathbb{E}_{comp}(m_1, v_1, v_2)$. Therefore, we express $\mathbb{E}_{verif}(m_1, v_2)$ as follows:

$$\mathbb{E}_{verif}(m_1, v_2) = \min_{m_1 \leq v_1 < v_2} \left\{ \mathbb{E}_{verif}(m_1, v_1) + \mathbb{E}_{comp}(m_1, v_1, v_2) \right\} , \tag{4.1}$$

which is initialized by:

$$\mathbb{E}_{verif}(m_1, m_1) = 0 ,$$

since there is no task to execute and no room for additional guaranteed verifications. Finally, note that we omit the cost of the guaranteed verification after T_{v_2} here, because it is accounted for in the function $\mathbb{E}_{comp}(m_1, v_1, v_2)$.

Computing $\mathbb{E}_{comp}(m_1, v_1, v_2)$

In order to compute the expected time needed to successfully execute tasks between two verifications at v_1 and v_2, we also need to remember the position of the last memory checkpoint at m_1.

First, we pay $W_{]v_1,v_2]}$, which is the time needed to execute tasks from T_{v_1+1} to T_{v_2}, and we account for the cost of the guaranteed verification V^*. Then, there is a probability $p^s_{]v_1,v_2]} = 1 - e^{-\lambda_s W_{]v_1,v_2]}}$ of detecting a silent error, in which case we recover from the last memory checkpoint with a cost R^M (set to 0 if $m_1 = 0$) and only re-execute the tasks from there, using the already computed $\mathbb{E}_{verif}(m_1, v_1)$ followed by $\mathbb{E}_{comp}(m_1, v_1, v_2)$ as before. Therefore, we obtain:

$$\mathbb{E}_{comp}(m_1, v_1, v_2) = W_{]v_1,v_2]} + V^* + p^s_{]v_1,v_2]} \left(R^M + \mathbb{E}_{verif}(m_1, v_1) + \mathbb{E}_{comp}(m_1, v_1, v_2)\right) .$$

Simplifying the equation above and solving for $\mathbb{E}_{comp}(m_1, v_1, v_2)$, we obtain:

$$\begin{aligned}\mathbb{E}_{comp}(m_1, v_1, v_2) = {} & e^{\lambda_s W_{]v_1,v_2]}} \left(W_{]v_1,v_2]} + V^*\right) \\ & + \left(e^{\lambda_s W_{]v_1,v_2]}} - 1\right) \left(R^M + \mathbb{E}_{verif}(m_1, v_1)\right) .\end{aligned}$$

Complexity

The complexity is now dominated by the computation of the 2D table $\mathbb{E}_{verif}(m_1, v_2)$, which contains $O(n^2)$ entries, and each entry depends on at most n other entries that are already computed. Hence, the overall complexity of the algorithm is $O(n^3)$ in time and $O(n^2)$ in space.

4.2.4 With partial verifications

It may be beneficial to further add partial verifications between two guaranteed verifications. The intuitive idea would be to add yet another level to the dynamic programming algorithm, and to replace $\mathbb{E}_{comp}(m_1, v_1, v_2)$ in Equation (4.1) by a call to a function $\mathbb{E}^{(intuitive)}_{partial}(m_1, v_1, p_2, v_2)$, with $p_2 = v_2$, which would compute the optimal expected time needed to execute all the tasks from T_{v_1+1} to T_{p_2} successfully (accounting for errors and re-executions) and add further partial verifications between v_1 and p_2.

However, while the dynamic programming approach was rather intuitive with guaranteed verifications, the problem becomes much harder when partial verifications come into play. Indeed, while computing $\mathbb{E}^{(intuitive)}_{partial}(m_1, v_1, p_2, v_2)$, there is a probability g that an error remains undetected after p_2. When this happens, we need to account for the time lost executing the following tasks until the error is eventually detected by the subsequent partial verifications, or in the worst case by the guaranteed verification at v_2. This is only possible if we know the optimal positions of the partial verifications between p_2 and v_2. This requires the dynamic programming algorithm to first compute the values on the right of p_2, hence progressing the opposite way as what has been done so far.

In other words, instead of calling $\mathbb{E}^{(intuitive)}_{partial}(m_1, v_1, p_2, v_2)$ (with $p_2 = v_2$) on the first call to place additional partial verifications between v_1 and p_2 (on the left of p_2), we now call $\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$ (with $p_1 = v_1$ on the first call) to place additional partial verifications between p_1 and v_2 (on the right of p_1), as illustrated in Figure 4.4. $\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$ is then the optimal expected time needed to execute all the tasks from T_{p_1+1} to T_{v_2}, where T_{p_1} is followed by a partial verification (with the exception of the first call where $p_1 = v_1$) and T_{v_2} is followed by a guaranteed verification, knowing the position of the last memory checkpoint at m_1 and the position of the last guaranteed verification at v_1.

However, this generates another problem: when an error occurs between p_1 and v_2, we need to account for the time lost re-executing all tasks between m_1 and v_1 (which is already computed in $\mathbb{E}_{verif}(m_1, v_1)$) and from v_1 to p_1, which is only possible if we know the optimal positions of the partial verifications between v_1 and p_1 (on the left of p_1). Since we are now solving the sub-problems on the right of p_1 first, we do not know these locations yet.

There is still hope. We make the following simple observation about the re-execution cost: if an error occurs between p_1 and v_2, or occurs earlier and is not detected by p_1, we will re-execute tasks between m_1 and v_1, and between v_1 and p_1, *regardless* of the number and positions of partial verifications between p_1 and v_2. Indeed, the error will be detected either sooner by a partial verification, or later by the guaranteed verification after T_{v_2}. In other words, the expected number of times that tasks between v_1 and p_1 are re-executed due to errors between p_1 and v_2 does not depend on the number and positions of partial verifications between p_1 and v_2: the only thing that matters is that the error will be detected eventually, and the task will be re-executed.

This leads us to the following approach: while deciding the optimal positions of partial verifications between p_1 and v_2, we can ignore the re-execution cost of tasks between v_1 and p_1: these tasks will be re-executed the same number of times in expectation, regardless of the decision we make here, and the total re-execution cost due to errors between p_1 and v_2 is only affected by the positions of p_1 and v_2. Therefore, we account for this cost later, while deciding the optimal position of p_2 between p_1 and v_2, knowing how many times we will execute (and re-execute) tasks between p_1 and p_2 in total (see Lemma 9 for additional details), effectively making $\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$ an independent subproblem.

Altogether, the following theorem presents a sophisticated dynamic programming algorithm when using partial verifications:

Theorem 14. *The optimal solution to the* SILENT *problem while using partial verifications can be obtained using a dynamic programming algorithm in $O(n^5)$ time and $O(n^4)$ space, where n is the number of tasks in the chain.*

The remainder of this section is devoted to proving this theorem. The first two levels of this dynamic programming algorithm, i.e., placing memory checkpoints and guaranteed verifications, are exactly the same as the ones presented in Theorem 13, except that we replace the call to $\mathbb{E}_{comp}(m_1, v_1, v_2)$ by a call to $\mathbb{E}_{partial}(m_1, v_1, v_1, v_2)$, as we did before with guaranteed verifications between memory checkpoints (see Section 4.2.3). The following describes the additional steps required in order to place partial verifications.

Placing partial verifications

Let $\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$ denote the optimal expected time needed to execute all the tasks from T_{p_1+1} to T_{v_2}, where T_{p_1} is followed by a partial verification (with the exception of the first call where $p_1 = v_1$) and T_{v_2} is followed by a guaranteed verification, knowing the position of the last memory checkpoint at m_1 and the position of the last guaranteed verification at v_1.

This is yet another level of dynamic programming: suppose that $m_1 = v_1 = 0$ and $v_2 = n$, the goal is to compute $\mathbb{E}_{partial}(0, 0, n, n), \mathbb{E}_{partial}(0, 0, n-1, n), \ldots, \mathbb{E}_{partial}(0, 0, 0, n)$, in this particular order (which is mandatory as shown below in Section 4.2.4) so that at each step, we

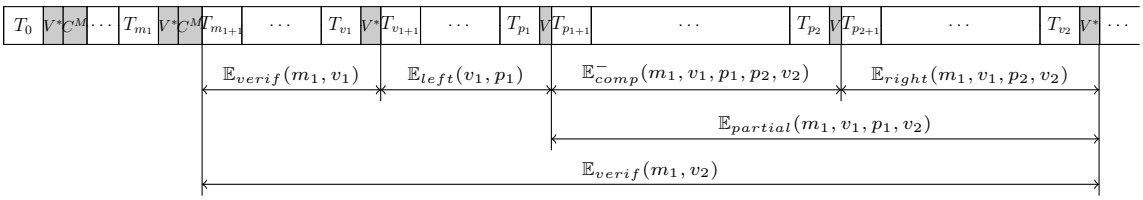


Figure 4.4: Placing partial verifications: m_1, v_1, p_1 and v_2 are fixed, and we try all possible locations for p_2. Here, both $\mathbb{E}_{verif}(m_1, v_1)$ and $\mathbb{E}_{right}(m_1, v_1, p_2, v_2)$, with $v_1 < p_2 \leq v_2$, have already been computed, which makes it possible to compute $\mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2)$, and then $\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$. Note that we do not need $\mathbb{E}_{left}(v_1, p_1)$ (see Section 4.2.4 and Lemma 9).

can reuse all previously computed optimal results. Clearly, we will have to solve this problem $O(n^3)$ times, for all possible $0 \leq m_1 \leq v_1 < v_2 \leq n$.

Similarly as what was done before, we compute $\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$ by deciding where to place an additional partial verification between tasks T_{p_1+1} and T_{v_2}, and we can write:

$$\mathbb{E}_{partial}(m_1, v_1, p_1, v_2) = \min_{p_1 < p_2 \leq v_2} \left\{ \mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2) + \mathbb{E}_{partial}(m_1, v_1, p_2, v_2) \right\},$$

where $\mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2)$ denotes the expected time needed to successfully execute all the tasks from T_{p_1+1} to T_{p_2}, and the initialization is:

$$\mathbb{E}_{partial}(m_1, v_1, v_2, v_2) = 0 \,,$$

because there is no task to execute and no more room for addition partial verifications.

Computing $\mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2)$

Recall that $\mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2)$ denotes the expected time needed to successfully execute all the tasks from T_{p_1+1} to T_{p_2}, accounting for the time lost due to errors and re-executions, with no undetected silent error after T_{p_2}, knowing that the last memory checkpoint is after T_{m_1}, the last guaranteed verification is after T_{v_1}, and the next guaranteed verification is after T_{v_2}. In order to compute $\mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2)$, we need to introduce $\mathbb{E}_{left}(v_1, p_1)$, the expected time needed to successfully execute tasks T_{v_1+1} to T_{p_1}, and $\mathbb{E}_{right}(m_1, v_1, p_2, v_2)$, the expected time lost executing the tasks following T_{p_2}, knowing that *there is an undetected silent error*. Note that in the worst scenario, a silent error will always be detected by the guaranteed verification after T_{v_2}.

Figure 4.4 shows all the tasks involved in the computation between two partial verifications at p_1 and p_2. Intuitively, the execution goes as follows. First, we execute all the tasks from T_{p_1+1} up to the next partial verification after T_{p_2}, and we pay $W_{]p_1,p_2]}$. Then, we add the cost V for the partial verification, and there is a probability $p^s_{]p_1,p_2]}$ of having a silent error. On the one hand, there is a probability $1-g$ to detect the error right after the partial verification at T_{p_2}. In this case, we pay a recovery cost R^M from the last memory checkpoint and re-execute the tasks from there by calling $\mathbb{E}_{verif}(m_1, v_1)$, followed by $\mathbb{E}_{left}(v_1, p_1)$ and $\mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2)$. However, if

the error is not detected (with probability g), we also account for the cost of re-executing tasks from the last memory checkpoint until T_{p_2}, but we further use $\mathbb{E}_{right}(m_1, v_1, p_2, v_2)$ to compute the expected time lost executing the tasks following T_{p_2}, knowing that there is an undetected silent error. In this case, the recovery cost will be accounted for in $\mathbb{E}_{right}$. Therefore, we have:

$$\begin{aligned}\mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2) = W_{]p_1,p_2]} + V + p^s_{]p_1,p_2]}\Big(&\mathbb{E}_{verif}(m_1, v_1)\\ &+ \mathbb{E}_{left}(v_1, p_1) + \mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2)\\ &+ (1-g)R^M + g\mathbb{E}_{right}(m_1, v_1, p_2, v_2)\Big)\,.\end{aligned}$$

Simplifying the equation above, we obtain:

$$\begin{aligned}\mathbb{E}_{comp}(m_1, v_1, p_1, p_2, v_2) = &\, e^{\lambda_s W_{]p_1,p_2]}}\left(W_{]p_1,p_2]} + V\right)\\ &+ \left(e^{\lambda_s W_{]p_1,p_2]}} - 1\right)\left(\mathbb{E}_{verif}(m_1, v_1) + \mathbb{E}_{left}(v_1, p_1)\right)\\ &+ \left(e^{\lambda_s W_{]p_1,p_2]}} - 1\right)\left((1-g)R^M + g\mathbb{E}_{right}(m_1, v_1, p_2, v_2)\right)\,. \qquad (4.2)\end{aligned}$$

Now, due to the order in which we solve the subproblems, both $\mathbb{E}_{verif}(m_1, v_1)$ and $\mathbb{E}_{partial}(m_1, v_1, p_2, v_2)$ have already been computed and we can get the optimal positions of partial verifications between T_{p_2+1} and T_{v_2}, which are needed to compute $\mathbb{E}_{right}(m_1, v_1, p_2, v_2)$. However, because we solve the subproblems the opposite way as what was done so far, we do not know (yet) the optimal positions of partial verifications between T_{v_1} and T_{p_1}, which are required to compute $\mathbb{E}_{left}(v_1, p_1)$.

Instead, we remove the term

$$\left(e^{\lambda_s W_{p_1,p_2}} - 1\right)\mathbb{E}_{left}(v_1, p_1)$$

from Equation (4.2), and introduce the modified expression of $\mathbb{E}_{comp}$, denoted by $\mathbb{E}^-_{comp}$, as follows:

$$\begin{aligned}\mathbb{E}^-_{comp}(m_1, v_1, p_1, p_2, v_2) = &\, e^{\lambda_s W_{]p_1,p_2]}}\left(W_{]p_1,p_2]} + V\right)\\ &+ \left(e^{\lambda_s W_{]p_1,p_2]}} - 1\right)\mathbb{E}_{verif}(m_1, v_1)\\ &+ \left(e^{\lambda_s W_{]p_1,p_2]}} - 1\right)\left((1-g)R^M + g\mathbb{E}_{right}(m_1, v_1, p_2, v_2)\right)\,. \qquad (4.3)\end{aligned}$$

Then, to account for the missing $\mathbb{E}_{left}$, we make use of Lemma 9, which shows that, for any number and position of partial verifications between T_{p_2+1} and T_{v_2}, $\mathbb{E}^-_{comp}(m_1, v_1, p_1, p_2, v_2)$ is executed $e^{\lambda_s W_{]p_2,v_2]}}$ times in expectation. As explained before, the intuition behind this result is that the amount of time tasks T_{v_1} to T_{p_1} will be re-executed due to errors in T_{p_1} to T_{v_2} does not depend upon the positions of intermediate partial verifications (e.g., p_2). For every error that occurs between T_{p_1} and T_{v_2}, these tasks *will* be re-executed *regardless* of the position of p_2 (or any other partial verifications between T_{p_1} and T_{v_2}). Hence, instead of accounting for the execution of T_{p_1} to T_{p_2} just once, we now account for all the times we have to execute, and re-execute them due to errors between T_{p_2} and T_{v_2}, and we obtain:

$$\mathbb{E}_{partial}(m_1, v_1, p_1, v_2) = \min_{p_1<p_2\le v_2}\left\{\mathbb{E}^-_{comp}(m_1, v_1, p_1, p_2, v_2)\cdot e^{\lambda_s W_{]p_2,v_2]}} + \mathbb{E}_{partial}(m_1, v_1, p_2, v_2)\right\}, \qquad (4.4)$$

which is initialized by:

$$\begin{aligned}
\mathbb{E}_{partial}(m_1, v_1, v_2, v_2) &= 0\ ; \\
\mathbb{E}^{-}_{comp}(m_1, v_1, p_1, v_2, v_2) &= e^{\lambda_s W_{]p_1,v_2]}} \left(W_{]p_1,v_2]} + V^*\right) \\
&\quad + \left(e^{\lambda_s W_{]p_1,v_2]}} - 1\right) \mathbb{E}_{verif}(m_1, v_1) \\
&\quad + \left(e^{\lambda_s W_{]p_1,v_2]}} - 1\right) R^M\ ,
\end{aligned}$$

because there is no task to execute and no more room for additional partial verifications, and because when computing $\mathbb{E}^{-}_{comp}(m_1, v_1, p_1, v_2, v_2)$ with $p_2 = v_2$, we need to account for the cost of the guaranteed verification V^* instead of the partial verification. Indeed, Lemma 9 (see below) proves that for any number of partial verifications between p_2 and v_2, $\mathbb{E}^{-}_{comp}(m_1, v_1, p_1, p_2, v_2)$ is executed $e^{\lambda_s W_{]p_2,v_2]}}$ times in expectation.

Computing $\mathbb{E}_{right}(m_1, v_1, p_1, v_2)$

Finally, in order to get $\mathbb{E}^{-}_{comp}(m_1, v_1, p_1, v_2, v_2)$, we still need compute $\mathbb{E}_{right}(m_1, v_1, p_1, v_2)$, the *optimal* expected time lost executing the tasks T_{p_1+1} to T_{v_2}, assuming that *there is an undetected silent error* in this interval. This computation uses p_2, the *optimal* position of the partial verification immediately following p_1, and it is computed by the dynamic programming algorithm. Indeed, the partial verification after T_{p_2} may or may not detect the error. If the error is detected, we lose $W_{]p_1,p_2]} + V + R^M$ time, while we use $\mathbb{E}_{right}(m_1, v_1, p_2, v_2)$ if the error remains undetected. Altogether, we have:

$$\mathbb{E}_{right}(m_1, v_1, p_1, v_2) = W_{]p_1,p_2]} + V + (1-g)R^M + g\mathbb{E}_{right}(m_1, v_1, p_2, v_2)\ , \tag{4.5}$$

where p_2 is the optimal position of the next partial verification knowing that there is a partial verification after T_{p_1} and is obtained by backtracking the last step as follows:

$$p_2 = \underset{p_1 < p_2 \leq v_2}{\arg\min} \left\{ \mathbb{E}^{-}_{comp}(m_1, v_1, p_1, p_2, v_2) \cdot e^{\lambda_s W_{]p_2,v_2]}} + \mathbb{E}_{partial}(m_1, v_1, p_2, v_2) \right\}\ .$$

Note that both $\mathbb{E}^{-}_{comp}(m_1, v_1, p_1, p_2, v_2)$, which only requires $\mathbb{E}_{right}(m_1, v_1, p_2, v_2)$ to be known, and $\mathbb{E}_{partial}(m_1, v_1, p_2, v_2)$ have already been computed by $\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$. In addition, recall that the time needed to re-execute the tasks after a recovery is not included here.

The initialization is as follows:

$$\begin{aligned}
&\mathbb{E}_{right}(m_1, v_1, v_2, v_2) = R^M, && \text{if } p_1 = p_2 = v_2 \\
&\mathbb{E}_{right}(m_1, v_1, p_1, v_2) = W_{]p_1,p_2]} + V^* + R^M, && \text{if } p_1 < p_2 \text{ and } p_2 = v_2
\end{aligned}$$

Indeed, when $p_2 = v_2$ there is no task to execute after T_{p_2}, and if there was a silent error, it is immediately detected at v_2 (by guaranteed verification), and we just pay R^M. Then, when $p_1 < p_2$ and $p_2 = v_2$, if an error goes undetected after p_1, we will execute tasks from T_{p_2} to T_{v_2}, and we need to take into account the cost of the guaranteed verification V^* at v_2. Again, the error is detected at v_2 and we pay R^M.

Complexity

Clearly, the complexity is now dominated by the computation of the table $\mathbb{E}_{partial}(m_1, v_1, p_1, v_2)$, which contains $O(n^4)$ entries, and each entry depends on at most n other entries that are already computed. Hence, the overall complexity of the algorithm is $O(n^5)$ in time and $O(n^4)$ in space.

Finally, the following presents the formalization and proof of Lemma 9, which has been used in proving Theorem 14.

Lemma 9. *For any number of partial verifications between p_2 and v_2, $\mathbb{E}^-_{comp}(m_1, v_1, p_1, p_2, v_2)$ is executed $e^{\lambda_s W_{]p_2,v_2]}}$ times in expectation.*

Proof. Looking at Equation (4.3), if there is no partial verification after p_2, then we must execute $\mathbb{E}^-_{comp}(m_1, v_1, p_1, p_2, v_2)$ at least once when progressing within the computation. Accounting for the term $\mathbb{E}_{left}$ that was suppressed from the final $\mathbb{E}_{comp}(m_1, v_1, p_2, v_2, v_2)$, we must re-execute $\mathbb{E}^-_{comp}(m_1, v_1, p_1, p_2, v_2)$ an additional $e^{\lambda_s W_{]p_2,v_2]}} - 1$ times due to errors occurring in $\mathbb{E}_{comp}(m_1, v_1, p_2, v_2, v_2)$. Overall, the expected number of times $\mathbb{E}^-_{comp}(m_1, v_1, p_1, p_2, v_2)$ is executed will be

$$1 + (e^{\lambda_s W_{]p_2,v_2]}} - 1) = e^{\lambda_s W_{]p_2,v_2]}} .$$

Now, with one intermediate partial verification p_3 between p_2 and v_2, the same reasoning shows that $\mathbb{E}^-_{comp}(m_1, v_1, p_2, p_3, v_2)$ must be executed $e^{\lambda_s W_{]p_3,v_2]}}$ times in expectation. Therefore, $\mathbb{E}^-_{comp}(m_1, v_1, p_1, p_2, v_2)$ must be executed once, coming from the initial execution, plus an additional $e^{\lambda_s W_{]p_2,p_3]}} - 1$ times due to the re-executions coming from the $\mathbb{E}_{left}$ term suppressed from $\mathbb{E}_{comp}(m_1, v_1, p_2, p_3, v_2)$, which is itself executed $e^{\lambda_s W_{]p_3,v_2]}}$ times. Finally, we must account for the $e^{\lambda_s W_{]p_3,v_2]}} - 1$ times coming from the last $\mathbb{E}_{comp}(m_1, v_1, p_3, v_2, v_2)$ as well. Overall, the expected number of times $\mathbb{E}^-_{comp}(m_1, v_1, p_1, p_2, v_2)$ is executed will be

$$1 + \left(e^{\lambda_s W_{]p_2,p_3]}} - 1\right) \cdot e^{\lambda_s W_{]p_3,v_2]}} + \left(e^{\lambda_s W_{]p_3,v_2]}} - 1\right) = e^{\lambda_s W_{]p_2,v_2]}} .$$

It is straightforward to extend this argument to any number of intervals by induction, assuming that it is true for i intermediate partial verifications $p_1, \ldots, p_i$, followed by a guaranteed verification, and adding a partial verification p_{i+1} between p_i and v_2. The same reasoning holds, which concludes the proof. □

4.3 Multi-level checkpointing for fail-stop errors

In this section, we present a multi-level dynamic programming algorithm to decide which tasks to checkpoint and at which levels when dealing with fail-stop errors. We first introduce the application and checkpointing models in Section 4.3.1 before presenting the dynamic programming algorithm in Section 4.3.2.

4.3.1 Model

As before, we consider a chain $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ of n tasks that execute on a large-scale platform subject to k levels of fail-stop errors. Recall that the weight w_i of task T_i is

known to the algorithm, and $W_{]i,j]} = \sum_{p=i+1}^{j} w_p$ denotes the total weight of tasks T_{i+1} to T_j for any $i < j$. Errors of different levels are assumed to be independent, and their arrivals follow *Poisson process* with error rate λ_ℓ for level ℓ, where $1 \leq \ell \leq k$. There are correspondingly k levels of checkpoints available, and each level ℓ is associated with a checkpointing cost $C^{(\ell)}$ and a recovery cost $R^{(\ell)}$. Typically, error rates are decreasing and checkpoint/recovery costs are increasing when we go to higher levels: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k$, $C^{(1)} \leq C^{(2)} \leq \cdots \leq C^{(k)}$, and $R^{(1)} \leq R^{(2)} \leq \cdots \leq R^{(k)}$. A level ℓ error destroys all the checkpoints of lower levels (from 1 to $\ell - 1$) and we need to roll back to a checkpoint of level ℓ or higher for recovery. Similarly, a recovery from a level ℓ checkpoint will restore data from all the lower levels. We assume that the costs of checkpointing and recovery are uniform across different tasks, and that they are protected from faults (i.e., errors only strike the computations).

For convenience, we add again before task T_1 a *virtual* task T_0, which is checkpointed at all levels, and whose checkpointing and recovery costs are always zero. This accounts for the fact that it is always possible to restart the application from scratch (i.e., recover from T_0) with no extra cost. Furthermore, we assume that the last task T_n is also always checkpointed at all levels in order to save the final outcome of the computation.

The MULTILEVEL problem consists in finding the optimal set of tasks that should be checkpointed at each level in order to minimize the total expected execution time of the task chain, accounting for failures and re-executions.

4.3.2 Dynamic programming algorithm

The main difference between fail-stop and silent error is the detection latency: fail-stop errors typically occur in the middle of the computation, causing the application to crash immediately and losing some data. From the algorithmic perspective, it has two implications: (i) when an error occurs, we only need to account for the time lost since the last available checkpoint; and (ii) we must recover from the right checkpoint level, which depends on the type of the error. Our approach for the MULTILEVEL problem remains similar to the algorithm presented in Section 4.2.2 by using dynamic programming:

Theorem 15. *The optimal solution to the* MULTILEVEL *problem can be obtained using a dynamic programming algorithm in $O(n^{k+1})$ time and $O(n^k)$ space, where n is the number of tasks in the chain, and k is the number of checkpointing levels available.*

The dynamic programming algorithm consists of k nested levels. The remainder of this section is devoted to proving this theorem by detailing how the k levels of checkpoints are placed.

Placing checkpoints at level k

We start with the highest and most expensive level k. Let $\mathbb{E}_{rec}^{(k)}(c_k)$ denote the optimal expected execution time to successfully execute all tasks from T_1 to T_{c_k} (included), where c_k denotes the index of a task whose output is saved with a level-k checkpoint. Intuitively, we want to obtain:

$$\mathbb{E}_{rec}^{(k)}(n),$$

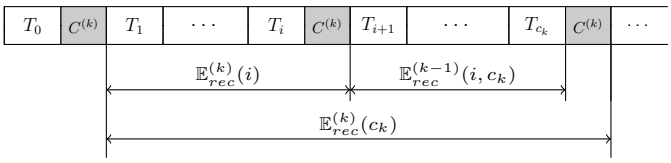


Figure 4.5: Placing checkpoints at level k: c_k is fixed, and we try all possible locations i for an additional checkpoint at level k between T_0 and T_{c_k}. Note that all subproblems $\mathbb{E}_{rec}^{(k)}(i)$, with $0 \leq i < c_k$, have already been computed, while $\mathbb{E}_{rec}^{(k-1)}(i, c_k)$ is computed by yet another dynamic programming level to be described later (see Figure 4.6).

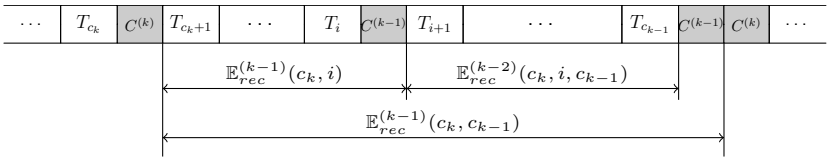


Figure 4.6: Placing checkpoints at level $k-1$: c_k and c_{k-1} are fixed, and we try possible locations i for an additional checkpoint between T_{c_k} and $T_{c_{k-1}}$. Again, all subproblems $\mathbb{E}_{rec}^{(k-1)}(c_k, i)$, with $c_k \leq i < c_{k-1}$, have already been computed, while $\mathbb{E}_{rec}^{(k-2)}(c_k, i, c_{k-1})$ is computed by the next level of dynamic programming level.

which is the optimal expected execution time to successfully execute all the tasks in the chain. Backtracking can then be used to get the corresponding optimal set of tasks to checkpoint.

In order to compute $\mathbb{E}_{rec}^{(k)}(c_k)$, we need to decide which tasks to checkpoint at level k between tasks T_0 and T_{c_k} (remember that T_0 is always checkpointed at level k). To this end, we consider each of these tasks as a potential candidate for the *last checkpoint* at level k before T_{c_k}, and return the minimum expected execution time as follows (see Figure 4.5):

$$\mathbb{E}_{rec}^{(k)}(c_k) = \min_{0 \leq i < c_k} \left\{ \mathbb{E}_{rec}^{(k)}(i) + \mathbb{E}_{rec}^{(k-1)}(i, c_k) \right\} + C^{(k)} .$$

For each task T_i, we first call $\mathbb{E}_{rec}^{(k)}(i)$ recursively to decide which additional tasks should be checkpointed at level k, between task T_1 and the newly checkpointed task T_i. Then, we compute the expected execution time between tasks T_i and T_{c_k} by calling the next level function $\mathbb{E}_{rec}^{(k-1)}(i, c_k)$ that decides which tasks to checkpoint at level $k-1$, knowing that both tasks T_i and T_{c_k} are already checkpointed at level k. Finally, we account for the level-k checkpointing cost $C^{(k)}$ after task T_{c_k}.

Placing checkpoints at level $k-1$.

Now, let $\mathbb{E}_{rec}^{(k-1)}(c_k, c_{k-1})$ denote the optimal expected execution time needed to successfully execute all the tasks from T_{c_k} to $T_{c_{k-1}}$ (included), where c_k denotes the position of the last checkpoint at level k, and c_{k-1} denotes the position of the next level $k-1$ checkpoint. Note that the first time we call this function (while computing $\mathbb{E}_{rec}^{(k)}(c_k)$ above), c_{k-1} (a.k.a. c_k above) actually denotes the position the next level-k checkpoint, which by construction always includes a

level $k-1$ checkpoint as well, and that is accounted for in the equation below. Similarly to the level-k function, we try all tasks T_i between T_{c_k} and $T_{c_{k-1}}$ (included) for the last checkpoint at level $k-1$, so that we can write (see Figure 4.6):

$$\mathbb{E}_{rec}^{(k-1)}(c_k, c_{k-1}) = \min_{c_k \leq i < c_{k-1}} \left\{ \mathbb{E}_{rec}^{(k-1)}(c_k, i) + \mathbb{E}_{rec}^{(k-2)}(c_k, i, c_{k-1}) \right\} + C^{(k-1)} .$$

We first call $\mathbb{E}_{rec}^{(k-1)}(c_k, i)$ recursively between tasks T_{c_k} and T_i to place additional checkpoints at level $k-1$, then call the function $\mathbb{E}_{rec}^{(k-2)}(c_k, i, c_{k-1})$ to place level $k-2$ checkpoints between tasks T_i and $T_{c_{k-1}}$, and finally account for the level $k-1$ checkpointing cost $C^{(k-1)}$ after task $T_{c_{k-1}}$. Note that T_{c_k} will always be checkpointed at level $k-1$, in addition of the level-k checkpoint that was already placed before. In fact, by following this approach, we guarantee that a high-level checkpoint always includes all the lower-level checkpoints as well.

Placing checkpoints at level ℓ

The function for placing checkpoints at level $k-2$ would now contain three parameters, because we need to remember both c_k and c_{k-1}, the positions of the last checkpoint at level k and level $k-1$, respectively, as well as c_{k-2}, the position of the next checkpoint at the current level $k-2$. This is because in case an error from level k or level $k-1$ strikes, we need to know which is the nearest available checkpoint to recover from.

In general, let $\mathbb{E}_{rec}^{(\ell)}(c_k, \ldots, c_{\ell+1}, c_\ell)$ denote the optimal expected execution time needed to execute tasks $T_{c_{\ell+1}+1}$ to T_{c_ℓ} (included), where $c_{\ell+1}$ is the position of the last checkpoint at level $\ell+1$ and c_ℓ is the position of the next level-ℓ checkpoint. Similarly to the functions at level k and level $k-1$, the goal of this function is to place additional level-ℓ checkpoints between tasks $T_{c_{\ell+1}}$ and T_{c_ℓ}. We denote by i the position of the newly added checkpoint at current level ℓ, and we try all possible positions between $c_{\ell+1}$ and c_ℓ. Hence, we derive:

$$\mathbb{E}_{rec}^{(\ell)}(c_k, \ldots, c_{\ell+1}, c_\ell) = \min_{c_{\ell+1} \leq i < c_\ell} \left\{ \mathbb{E}_{rec}^{(\ell)}(c_k, \ldots, c_{\ell+1}, i) + \mathbb{E}_{rec}^{(\ell-1)}(c_k, \ldots, c_{\ell+1}, i, c_\ell) \right\} + C^{(\ell)} . \tag{4.6}$$

For each candidate T_i, we first call $\mathbb{E}_{rec}^{(\ell)}(c_k, \ldots, c_{\ell+1}, i)$ to place additional level-ℓ checkpoints between tasks $T_{c_{\ell+1}}$ and T_i. Then, we call $\mathbb{E}_{rec}^{(\ell-1)}(c_k, \ldots, c_{\ell+1}, i, c_\ell)$ to place additional level $\ell-1$ checkpoints between tasks T_i and T_{c_ℓ}. Finally, we account for the level-ℓ checkpointing cost $C^{(\ell)}$ after task T_{c_ℓ}.

Initialization

To initialize the dynamic program at each level ℓ, we set:

$$\mathbb{E}_{rec}^{(\ell)}(c_k, \ldots, c_{\ell+1}, c_{\ell+1}) = 0 ,$$

which occurs once when $i = c_{\ell+1}$ in Equation (4.6); there is no task to execute, and the cost of the checkpoint after T_i has been accounted for already. Then, when the last level is reached, i.e., when $\ell = 1$, there is no more checkpointing level to try, and we set:

$$\mathbb{E}_{rec}^{(0)}(c_k, \ldots, c_2, c_1, c_{1'}) = \mathbb{E}_{comp}(c_k, \ldots, c_2, c_1, c_{1'}) ,$$

where $\mathbb{E}_{comp}(c_k, \ldots, c_2, c_1, c_{1'})$ denotes the expected execution time needed to execute tasks T_{c_1+1} to $T_{c_{1'}}$ (included), with no additional intermediate checkpoints in between.

Computing $\mathbb{E}_{comp}(c_k, \ldots, c_1, c_{1'})$

Given the positions of the checkpoints, we can now compute the actual expected execution time needed to successfully execute the tasks between any two consecutive level-1 checkpoints. We make use of the following well-known properties of independent Poisson processes [55, Chapter 2.3].

Property 2. *During the execution of a sequence of tasks with total work W, let X_ℓ denote the time when the first level-ℓ error strikes. Thus, X_ℓ is a random variable following* exponential *distribution with parameter λ_ℓ, for all $\ell = 1, 2, \ldots, k$.*

(1) Let X denote the time when the first error (of any level) strikes. We have $X = \min\{X_1, X_2, \ldots, X_k\}$, which follows exponential distribution with parameter $\Lambda = \sum_{\ell=1}^{k} \lambda_\ell$. The probability of having an error (from any level) during the execution is therefore $P(X \leq W) = 1 - e^{-\Lambda W}$.

(2) Given that an error (from any level) strikes during the execution of the tasks, the probability that the error belongs to a particular level is proportional to the error rate of that level, i.e., $P(X = X_\ell | X \leq W) = \frac{\lambda_\ell}{\Lambda}$, for all $\ell = 1, 2, \ldots, k$.

Recall that $W_{]c_1,c_{1'}]} = \sum_{i=c_1+1}^{c_{1'}} w_i$ denotes the total computational load between tasks T_{c_1+1} and $T_{c_{1'}}$. Hence, with probability $p^f_{]c_1,c_{1'}]} = 1 - e^{-\Lambda \cdot W_{]c_1,c_{1'}]}}$, at least one fail-stop error (from any level) *will* occur during the execution of tasks T_{c_1+1} to $T_{c_{1'}}$ (included). When this happens, we first need to account for the time lost during the execution (up to the error), denoted by $T^{\text{lost}}_{]c_1,c_{1'}]}$. Then, we need to roll back to the nearest checkpoint, depending on the level of the error. For example, with probability $\frac{\lambda_3}{\Lambda}$, we need to recover from the last level-3 checkpoint, and we pay $R^{(3)}$, the cost to recover from task T_{c_3} using level-3 recovery. When the recovery is done, we need to re-execute all the tasks, first from T_{c_3+1} to T_{c_2}, then from T_{c_2+1} to T_{c_1}, and finally from T_{c_1} to $T_{c_{1'}}$ again. By construction, there is no other level-3 checkpoint between T_{c_3} and T_{c_2}, so the expected time to re-execute all the tasks up to the next level-2 checkpoint, that is from tasks T_{c_3} to T_{c_2}, is simply $\mathbb{E}^{(2)}_{rec}(c_k, \ldots, c_3, c_2)$. Then, we proceed by re-executing the tasks up to the next level-1 checkpoint, which takes $\mathbb{E}^{(1)}_{rec}(c_k, \ldots, c_2, c_1)$ time, at which point we can just call $\mathbb{E}_{comp}(c_1, \ldots, c_1, c_{1'})$ again, to restart this whole process until the execution of tasks T_{c_1} to $T_{c_{1'}}$ is eventually successful. When no error occurs, which happens with probability $1 - p^f_{]c_1,c_{1'}]}$, we just need to pay the cost of executing all the tasks without error, i.e., $W_{]c_1,c_{1'}]}$. Therefore, we derive:

$$\mathbb{E}_{comp}(c_k,\dots,c_1,c_{1'}) = \left(1-p^f_{]c_1,c_{1'}]}\right) W_{]c_1,c_{1'}]} + p^f_{]c_1,c_{1'}]}\Bigg(T^{\text{lost}}_{]c_1,c_{1'}]}$$
$$+\frac{\lambda_1}{\Lambda}\Big(R^{(1)}+\mathbb{E}_{comp}(c_k,\dots,c_1,c_{1'})\Big)$$
$$+\frac{\lambda_2}{\Lambda}\Big(R^{(2)}+\mathbb{E}^{(1)}_{rec}(c_k,\dots,c_2,c_1)+\mathbb{E}_{comp}(c_k,\dots,c_1,c_{1'})\Big)$$
$$\vdots$$
$$+\frac{\lambda_{k-1}}{\Lambda}\Big(R^{(k-1)}+\sum_{\ell=2}^{k-1}\mathbb{E}^{(\ell-1)}_{rec}(c_k,\dots,c_\ell,c_{\ell-1})+\mathbb{E}_{comp}(c_k,\dots,c_1,c_{1'})\Big)$$
$$+\frac{\lambda_k}{\Lambda}\Big(R^{(k)}+\sum_{\ell=2}^{k}\mathbb{E}^{(\ell-1)}_{rec}(c_k,\dots,c_\ell,c_{\ell-1})+\mathbb{E}_{comp}(c_k,\dots,c_1,c_{1'})\Big)\Bigg)\,.$$

Simplifying the equation above, we can obtain:

$$\mathbb{E}_{comp}(c_k,\dots,c_1,c_{1'}) = e^{-\Lambda\cdot W_{]c_1,c_{1'}]}}W_{]c_1,c_{1'}]} + \left(1-e^{-\Lambda\cdot W_{]c_1,c_{1'}]}}\right)\Bigg(T^{\text{lost}}_{]c_1,c_{1'}]}$$
$$+\sum_{h=1}^{k}\frac{\lambda_h}{\Lambda}\Big(R^{(h)}+\sum_{\ell=2}^{h}\mathbb{E}^{(\ell-1)}_{rec}(c_k,\dots,c_\ell,c_{\ell-1})\Big)$$
$$+\mathbb{E}_{comp}(c_k,\dots,c_1,c_{1'})\Bigg)\,. \tag{4.7}$$

In order to compute the expected execution time, we need to compute $T^{\text{lost}}_{]c_1,c_{1'}]}$, which is the expected time loss due to a fail-stop error occurring during the execution of tasks T_{c_1} to $T_{c_{1'}}$. We obtain:

$$T^{\text{lost}}_{]c_1,c_{1'}]} = \int_0^\infty x\mathbb{P}(X=x|X<W_{]c_1,c_{1'}]})dx$$
$$= \frac{1}{\mathbb{P}(X<W_{]c_1,c_{1'}]})}\int_0^{W_{]c_1,c_{1'}]}} x\mathbb{P}(X=x)dx\,,$$

where $\mathbb{P}(X=x)$ denotes the probability that a fail-stop error strikes at time x. By definition, we have $\mathbb{P}(X=x)=\Lambda e^{-\Lambda x}$ and $\mathbb{P}(X<W_{]c_1,c_{1'}]})=1-e^{-\Lambda W_{]c_1,c_{1'}]}}$. Integrating by parts, we have:

$$T^{\text{lost}}_{]c_1,c_{1'}]} = \frac{1}{\Lambda}-\frac{W_{]c_1,c_{1'}]}}{e^{\Lambda W_{]c_1,c_{1'}]}}-1}\,. \tag{4.8}$$

Now, substituting $T^{\text{lost}}_{]c_1,c_{1'}]}$ above into Equation (4.7), and solving for $\mathbb{E}_{comp}(c_k,\dots,c_1,c_{1'})$, we obtain:

$$\mathbb{E}_{comp}(c_k,\dots,c_1,c_{1'}) = \frac{e^{\Lambda\cdot W_{]c_1,c_{1'}]}}-1}{\Lambda}\left(1+\sum_{h=1}^{k}\lambda_h\Big(R^{(h)}+\sum_{\ell=2}^{h}\mathbb{E}^{(\ell-1)}_{rec}(c_k,\dots,c_\ell,c_{\ell-1})\Big)\right)\,.$$

Complexity

The complexity is dominated by the computation of the table $\mathbb{E}^{(1)}_{rec}(c_k, \ldots, c_2, c_1)$, which contains n^k entries. In order to compute each entry, a minimum over at most n other entries (that are already computed) is required. All tables are computed in a bottom-up fashion, from the left to the right of the intervals. Hence, the overall complexity of the algorithm is $O(n^{k+1})$.

4.4 Dealing with both fail-stop and silent errors

On real-life platforms, fail-stop errors and silent errors coexist, and thus resilience algorithms must be able to cope with both error sources simultaneously. In this section, we describe a multi-level dynamic programming algorithm to address this challenging problem.

The new algorithm is a combination of the dynamic programming algorithms presented in the preceding sections. In particular, we place k levels of *disk*[2] checkpoints to deal with different fail-stop errors, followed by another level of memory checkpoints, and additional verifications (guaranteed or partial), to deal with silent errors. We call this problem the MULTILEVEL-SILENT problem, and the objective is to find the optimal positions in the task chain to place different checkpoints (disk and memory) as well as verifications (guaranteed and partial) to minimize the expected execution time. The following theorem presents the solution to this problem.

Theorem 16. *The optimal solution to the* MULTILEVEL-SILENT *problem can be obtained using a dynamic programming algorithm in $O(n^{k+5})$ time and $O(n^{k+4})$ space, where n is the number of tasks in the chain and k is number of checkpointing levels to handle fail-stop errors.*

Proof. The dynamic programming for fail-stop errors is exactly the same as the one shown in Section 4.3.2, up to the call to the function $\mathbb{E}^{(0)}_{rec}(c_k, \ldots, c_2, c_1, c_{1'})$, which is invoked after placing the last level-1 checkpoints. Now, in order to handle silent errors, we set:

$$\mathbb{E}^{(0)}_{rec}(c_k, \ldots, c_2, c_1, c_{1'}) = \mathbb{E}_{mem}(c_k, \ldots, c_1, c_{1'}) \ ,$$

where

$$\mathbb{E}_{mem}(c_k, \ldots, c_1, m_2) = \min_{c_1 \leq m_1 < m_2} \left\{ \mathbb{E}_{mem}(c_k, \ldots, c_1, m_1) + \mathbb{E}_{verif}(c_k, \ldots, c_1, m_1, m_2) \right\} + C^M \ ,$$

with $m_2 = c_{1'}$ when first called from $\mathbb{E}_{mem}(c_k, \ldots, c_1, m_2)$, and

$$\mathbb{E}_{verif}(c_k, \ldots, c_1, m_1, v_2) = \min_{m_1 \leq v_1 < v_2} \left\{ \mathbb{E}_{verif}(c_k, \ldots, c_1, m_1, v_1) + \mathbb{E}_{partial}(c_k, \ldots, c_1, m_1, v_1, v_2, v_2) \right\} \ ,$$

with $v_2 = m_2$ when first called from $\mathbb{E}_{verif}(c_k, \ldots, c_1, m_1, v_2)$. Overall, $\mathbb{E}_{mem}(c_k, \ldots, c_1, m_2)$ and $\mathbb{E}_{verif}(c_k, \ldots, c_1, m_1, v_2)$ remain the same as in Section 4.2.4, except for the fact that we now need to remember the position of the last checkpoint at each level, in case a fail-stop error occurs during the execution of tasks T_{v_1+1} to T_{v_2}. As for the initialization, we set

[2] By *disk*, we mean stable storage devices or advanced checkpointing mechanisms (e.g., partner-copy [12]) that can survive various sources of fail-stop errors, in opposition to *memory*, which can only survive silent errors.

$\mathbb{E}_{mem}(c_k, \dots, c_1, c_1) = 0$ and $\mathbb{E}_{verif}(c_k, \dots, c_1, m_1, m_1) = 0$, which occur once when $m_1 = c_1$ and $v_1 = m_1$, respectively. In both cases, there is no task to execute, and the cost of the checkpoint/verification has already been accounted for.

Placing partial verifications. Similarly, the function to place additional partial verifications becomes $\mathbb{E}_{partial}(c_k, \dots, c_1, m_1, v_1, p_1, v_2)$, and the expected number of times the function $\mathbb{E}^{-}_{comp}(c_k, \dots, c_1, m_1$ is executed must now account for both silent errors and fail-stop errors. Hence, we can write:

$$\mathbb{E}_{partial}(c_k, \dots, c_1, m_1, v_1, p_1, v_2) = \min_{p_1 < p_2 \leq v_2} \Big\{ \mathbb{E}^{-}_{comp}(c_k, \dots, c_1, m_1, v_1, p_1, p_2, v_2) \cdot e^{(\lambda_s + \Lambda) W_{]p_2, v_2]}} + \mathbb{E}_{partial}(c_k, \dots, c_1, m_1, v_1, p_2, v_2) \Big\} .$$

Computing $\mathbb{E}^{-}_{comp}(c_k, \dots, c_1, m_1, v_1, p_1, p_2, v_2)$. On the one hand, if a fail-stop error occurs with probability $p^f_{]p_1,p_2]} = 1 - e^{-\Lambda \cdot W_{]p_1,p_2]}}$, we can apply the same method as in Section 4.3. We recover from the nearest checkpoint depending on the error level, and we re-execute all the tasks up to $\mathbb{E}^{(1)}_{rec}(c_k, \dots, c_1)$, then we call $\mathbb{E}_{mem}(c_k, \dots, c_1, m_1)$, followed by a call to the function $\mathbb{E}_{verif}(c_k, \dots, c_1, m_1, v_1)$ to account for the time needed to re-execute the tasks between the last memory checkpoint after T_{m_1} to the next guaranteed verification after T_{v_1}, and finally we are left with the remaining tasks between T_{v_1+1} and T_{p_1}, and we call $\mathbb{E}^{-}_{comp}(c_k, \dots, c_1, m_1, v_1, p_1, p_2, v_2)$ again. On the other hand, with probability $(1 - p^f_{p_1,p_2})$, there is no fail-stop error. In that case, we execute all the tasks from T_{p_1+1} to the next verification after T_{p_2}, as was done in Section 4.2.4. Overall, we can write:

$$\begin{aligned}
\mathbb{E}^{-}_{comp}(c_k, \dots, c_1, m_1, v_1, p_1, p_2, v_2) = \\
& p^f_{]p_1,p_2]} \Bigg(T^{\text{lost}}_{]p_1,p_2]} + \sum_{h=1}^{k} \frac{\lambda_h}{\Lambda} \Big(R^{(h)} + \sum_{\ell=2}^{h} \mathbb{E}^{(\ell-1)}_{rec}(c_k, \dots, c_\ell, c_{\ell-1}) \Big) \\
& \qquad + \mathbb{E}_{mem}(c_k, \dots, c_1, m_1) + \mathbb{E}_{verif}(c_k, \dots, c_1, m_1, v_1) \\
& \qquad + \mathbb{E}^{-}_{comp}(c_k, \dots, c_1, m_1, v_1, p_1, p_2, v_2) \Bigg) \\
& + \Big(1 - p^f_{]p_1,p_2]}\Big) \Bigg(W_{]p_1,p_2]} + V + p^s_{]p_1,p_2]} \Big(\mathbb{E}_{verif}(c_k, \dots, c_1, m_1, v_1) \\
& \qquad + \mathbb{E}^{-}_{comp}(c_k, \dots, c_1, m_1, v_1, p_1, p_2, v_2) \\
& \qquad + (1 - g) R^M + g \mathbb{E}_{right}(c_k, \dots, c_1, m_1, v_1, p_2, v_2) \Big) \Bigg) .
\end{aligned}$$

Simplifying the equation above and solving for $\mathbb{E}^{-}_{comp}$, we obtain:

$$\begin{aligned}
\mathbb{E}^{-}_{comp}(c_k, \ldots, c_1, m_1, v_1, p_1, p_2, v_2) = \\
&+ e^{\lambda_s W_{]p_1,p_2]}} \left(\frac{e^{\Lambda W_{]p_1,p_2]}} - 1}{\Lambda} + V \right) \\
&+ e^{\lambda_s W_{]p_1,p_2]}} \left(e^{\Lambda W_{]p_1,p_2]}} - 1 \right) \left(\sum_{h=1}^{k} \frac{\lambda_h}{\Lambda} \Big(R^{(h)} + \sum_{\ell=2}^{h} \mathbb{E}^{(\ell-1)}_{rec}(c_k, \ldots, c_\ell, c_{\ell-1}) \Big) \right. \\
&\qquad \left. + \mathbb{E}_{mem}(c_k, \ldots, c_1, m_1) \right) \\
&+ \left(e^{(\lambda_s+\Lambda) W_{]p_1,p_2]}} - 1 \right) \mathbb{E}_{verif}(c_k, \ldots, c_1, m_1, v_1) \\
&+ \left(e^{\lambda_s W_{]p_1,p_2]}} - 1 \right) \Big((1-g) R^M + g \mathbb{E}_{right}(c_k, \ldots, c_1, m_1, v_1, p_2, v_2) \Big) \, .
\end{aligned}$$

Computing $\mathbb{E}_{right}(c_k, \ldots, c_1, m_1, v_1, p_1, v_2)$. Remember that $\mathbb{E}_{right}(c_k, \ldots, c_1, m_1, v_1, p_1, v_2)$ denotes the expected time lost executing the tasks T_{p_1+1} to T_{v_2}, assuming that *there was a silent error* in this interval. Equation (4.5) already accounts for the time lost in that case, but only when there is no fail-stop error. Similarly to $\mathbb{E}^{-}_{comp}$ above, we consider fail-stop errors between T_{p_1+1} and T_{p_2}, because fail-stop errors between T_{p_2+1} and T_{v_2} will be accounted for in $\mathbb{E}_{right}(c_k, \ldots, c_1, m_1, v_1, p_2, v_2)$. Note that even if we know that there is a silent error in the interval, we may need to recover from a fail-stop error if it strikes before the silent error is detected. Altogether, we have:

$$\begin{aligned}
\mathbb{E}_{right}(c_k, \ldots, c_1, m_1, v_1, p_1, v_2) = \\
&p^{f}_{]p_1,p_2]} \left(T^{\text{lost}}_{]p_1,p_2]} + \sum_{h=1}^{k} \frac{\lambda_h}{\Lambda} \Big(R^{(h)} + \sum_{\ell=2}^{h} \mathbb{E}^{(\ell-1)}_{rec}(c_k, \ldots, c_\ell, c_{\ell-1}) \Big) + \mathbb{E}_{mem}(c_k, \ldots, c_1, m_1) \right) \\
&+ \left(1 - p^{f}_{]p_1,p_2]} \right) \Big(W_{]p_1,p_2]} + V + (1-g) R^M + g \mathbb{E}_{right}(c_k, \ldots, c_1, m_1, v_1, p_2, v_2) \Big) \, .
\end{aligned}$$

Finally, simplifying the equation above, we obtain:

$$\begin{aligned}
\mathbb{E}_{right}(c_k, \ldots, c_1, m_1, v_1, p_1, v_2) = \\
&\left(1 - e^{-\Lambda W_{]p_1,p_2]}} \right) \left(\frac{1}{\Lambda} + \sum_{h=1}^{k} \frac{\lambda_h}{\Lambda} \Big(R^{(h)} + \sum_{\ell=2}^{h} \mathbb{E}^{(\ell-1)}_{rec}(c_k, \ldots, c_\ell, c_{\ell-1}) \Big) + \mathbb{E}_{mem}(c_k, \ldots, c_1, m_1) \right) \\
&+ e^{-\Lambda W_{]p_1,p_2]}} \Big(V + (1-g) R^M + g \mathbb{E}_{right}(c_k, \ldots, c_1, m_1, v_1, p_2, v_2) \Big) \, .
\end{aligned}$$

The initialization remains $\mathbb{E}_{right}(c_k, \ldots, c_1, m_1, v_1, v_2, v_2) = R^M$.

Complexity. The complexity is dominated by the computation of the dynamic programming table $\mathbb{E}_{partial}(c_k, \ldots, c_1, m_1, v_1, p_1, v_2)$, which contains $O(n^{k+4})$ entries, and each entry depends on at most n other entries that are already computed. Therefore, the complexity of the dynamic programming algorithm to handle both fail-stop and silent errors is $O(n^{k+5})$. □

We point out that, in practical systems, the number of checkpointing levels k is generally quite small and rarely exceeds 3 or 4 [12, 74], while linear application workflows rarely exceed a few tens of tasks. Hence, our algorithm can be efficiently applied to these practical scenarios in reasonable time and space.

4.5 Performance evaluation

In this section, we conduct a set of simulations to assess the relative efficiency of our approach under practical scenarios. We instantiate the performance model with two different sets of realistic parameters obtained from the literature. The simulation code is publicly available at `http://graal.ens-lyon.fr/~yrobert/chainmultilevel.zip` for interested readers to experiment with their own parameters.

Simulation setup. We make several assumptions on the input parameters. First, the checkpoint and recovery costs both depend on size of the the task output file, and the final cost is mostly determined by the available bandwidth at each level. As such, we make the assumption that the recovery cost for a given level is equivalent to the corresponding checkpointing cost, i.e., $R^{(i)} = C^{(i)}$ for $1 \leq i \leq k$. This is a common assumption [37, 74, 79], even though in practice the recovery cost can be expected to be smaller than the checkpoint cost [37, 39].

Then, we assume that, similarly, a guaranteed verification must check all the data in memory, making its cost in the same order as that of a memory checkpoint, i.e., $V^* = C^M$. Furthermore, we assume partial verifications similar to those proposed in [8, 10, 15], with very low costs while offering good recalls. In the following, we set $V = V^*/100$ and $r = 0.8$. The total computational weight is set to be $W = 25000$ seconds (or $W = 3600s$ in some simulations), and it is distributed among up to $n = 50$ tasks in three different patterns shown as follows.

(1) *Uniform*: all tasks share the same weight W/n, as in matrix multiplication or in some iterative stencil kernels.

(2) *Decrease*: task T_i has weight $\alpha(n + 1 - i)^2$, where $\alpha \approx 3W/n^3$; this quadratically decreasing function resembles some dense matrix solvers, e.g., by using LU or QR factorization.

(3) *HighLow*: a set of tasks with large weight is followed by a set of tasks with small weight. In the simulations, we set 10% of the tasks to be large and let them contain 60% of the total computational weight.

We point out that all these choices are somewhat arbitrary and can easily be modified in the evaluations; however we believe they represent reasonable values for current and next-generation HPC applications. We first investigate the impact of using guaranteed and partial verifications in Section 4.5.1, by focusing on a platform with a single level of checkpoints for fail-stop errors. Then, we study the impact of multi-level checkpointing in Section 4.5.2.

4.5.1 Results for two-level checkpointing

In this section, we perform a set of experiments based on the characteristics of four platforms taken from the literature. We start by analyzing the combined algorithm, but in a somewhat simplified context, with only one level of checkpoint to deal with fail-stop errors (i.e., $k = 1$). We compare three algorithms: (i) a single-level algorithm A_{DV^*} with only disk checkpoints to

handle both fail-stop and silent errors (with additional guaranteed verifications); (ii) a two-level algorithm A_{DMV^*} with additional memory checkpoints for silent errors; and (iii) the combined algorithm A_{DMV} using additional partial verifications. The optimal positions of verifications can be easily derived for A_{DV^*} using a simplification of the proposed dynamic programming algorithm in Section 4.4, with $k = 1$ level of fail-stop errors and no additional memory checkpoints.

Platform settings. Table I presents the four platforms used in the simulations and their main parameters. These platforms have been used to evaluate the Scalable Checkpoint/Restart (SCR) library by Moody et al. [74], who provide accurate measurements for λ_f, λ_s, $C^{(1)}$ and C^M using real applications. Note that in this configuration, $C^{(1)}$ denotes the cost of checkpointing to disk, and is referred to as a disk checkpoint below, as opposed to the memory checkpoint, which is done in RAM. There is an exception with the Coastal platform, on which SSD technology is used for memory checkpointing; this provides more data space, at the cost of higher checkpointing costs. In addition, note that the Hera platform has the worst error rates, with a platform MTBF of 12.2 days for fail-stop errors and 3.4 days for silent errors. In comparison, and despite its higher number of nodes, the Coastal platform features a platform MTBF of 28.8 days for fail-stop errors and 5.8 days for silent errors.

Set	From	Platform	#Nodes	λ_f	λ_s	$C^{(1)}$	C^M
(A)	Moody et al. [74]	Hera	256	9.46e-7	3.38e-6	$300s$	$15.4s$
		Atlas	512	5.19e-7	7.78e-6	$439s$	$9.1s$
		Coastal	1024	4.02e-7	2.01e-6	$1051s$	$4.5s$
		Coastal SSD	1024	4.02e-7	2.01e-6	$2500s$	$180.0s$

Table I
SET OF PARAMETERS (A) USED AS INPUT FOR SIMULATIONS.

Impact of the number of tasks. The first column of Figure 4.7 presents, for each platform, the normalized makespan with respect to the error-free execution time for different numbers of tasks with the *Uniform* pattern. Note that varying the number of tasks has an impact on both the size of the tasks and the maximum number of checkpoints and verifications that the scheduling algorithm can place. When the number of tasks is small (e.g., less than 5), the probability of having an error during the execution (either a fail-stop or a silent) increases quickly (more than 10% on Hera) for a single task. As a result, the application experiences more recoveries and re-executions with larger tasks, which increases the execution overhead. However, when the number of tasks is large enough, the size of the tasks becomes small and the probability of having an error during the execution of one task drops significantly, reducing the recovery and re-execution costs at the same time.

Single-level algorithm A_{DV^*}. The second column of Figure 4.7 shows the numbers of disk checkpoints (with associated memory checkpoints) and guaranteed verifications used by the A_{DV^*} algorithm on the four platforms and for different numbers of tasks. We observe that a large number of guaranteed verifications is placed by the algorithm while the number of checkpoints remains relatively small (e.g., less than 5 for all the platforms). This is because checkpoints are costly, and verifications help reduce the amount of time lost due to silent errors.

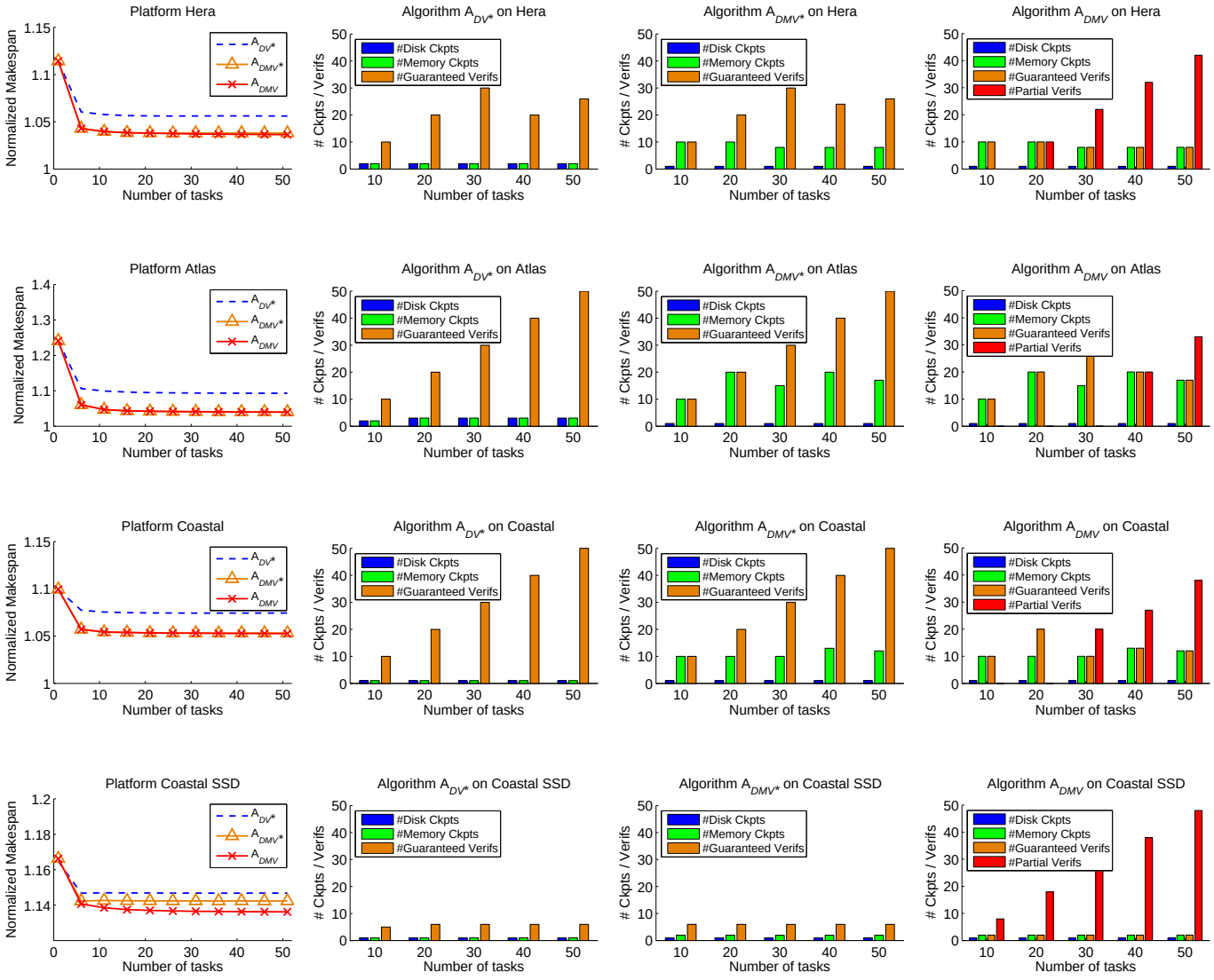


Figure 4.7: Performance of the three algorithms on each platform with the *Uniform* pattern. Each row corresponds to one platform.

Since verifications are cheaper, the algorithm tends to place as many of them as possible, except when their relative costs also become high (e.g., on Coastal SSD). In addition, when the number of tasks is large enough (e.g., $n > 30$ on Hera), not all tasks need to be verified. Note that there are fewer verifications for $n = 40$ than for $n = 30$, while the number of memory checkpoints remain the same. In fact, there is a threshold ($n = 38$) after which one verification for one task becomes an overkill. Instead, the algorithm places one verification every two tasks, resulting in exactly 20 guaranteed verifications (including the ones before each memory checkpoint) at $n = 40$ tasks.

Two-level algorithm $\mathbf{A}_{DMV^*}$. The third column of Figure 4.7 presents the numbers of disk checkpoints, memory checkpoints and guaranteed verifications used by the A_{DMV^*} algorithm on the four platforms and for different numbers of tasks. We observe that the number of guaranteed verifications remains similar to that placed by the A_{DV^*} algorithm. However, the two-level algorithm uses additional memory checkpoints, which drastically reduces the amount of time lost in re-execution when a silent error is detected. In particular, we observe that the algorithm A_{DMV^*} always leads to a better makespan compared to the single-level algorithm A_{DV^*}, with an improvement of 2% on Hera and 5% on Atlas, as shown in the first column of Figure 4.7. This demonstrates the usefulness of the multi-level checkpointing approach.

Combined algorithm $\mathbf{A}_{DMV}$. The last column of Figure 4.7 presents the numbers of disk checkpoints, memory checkpoints, guaranteed verifications and additional partial verifications used by the A_{DMV} algorithm on the four platforms and for different numbers of tasks. Although partial verifications are always more cost-effective than guaranteed ones, due to the imperfect recall, they are only useful if one can use a lot of them, which is only possible when the number of tasks is large enough. Therefore, the algorithm only starts to use partial verifications when the number of tasks is greater than 30 on Hera, 40 on Coastal and 50 on Atlas, where silent error rate is the highest among the four platforms. In our setting, adding partial verifications has a limited impact on the makespan, with the exception of the Coastal SSD platform, where the cost of checkpoints and verifications are much higher than on the other platforms. Partial verifications, being 100 times cheaper than guaranteed verifications, remain the only affordable resilience tool on this platform. In this case, we observe an improved makespan (around 1% with 50 tasks) compared to the A_{DMV^*} algorithm, as shown in the first column of Figure 4.7.

Distribution of checkpoints and verifications. Figure 4.8 shows the positions of the disk checkpoints, memory checkpoints, verifications and partial verifications obtained by running the A_{DMV} algorithm on each of the four platforms and for 50 tasks with the uniform distribution. For all platforms, the algorithm does not perform any additional disk checkpoints. These being costly, the algorithm rather uses more memory checkpoints and verifications. On most platforms, the optimal solution is a combination of equi-spaced memory checkpoints and guaranteed verifications, with additional partial verifications in-between. However, on the Coastal SSD platform, the cost of checkpoints and verifications is substantially higher, which leads the algorithm to choose partial verifications rather than guaranteed ones.

***Decrease* pattern.** In the following, we focus on the platforms Hera and Coastal SSD, which represent both extremes in terms of size (number of nodes) and hardware used for memory checkpointing (RAM and SSD, respectively). The first column of Figure 4.9 presents the performance of the three algorithms for different numbers of tasks and for the *Decrease* pattern.

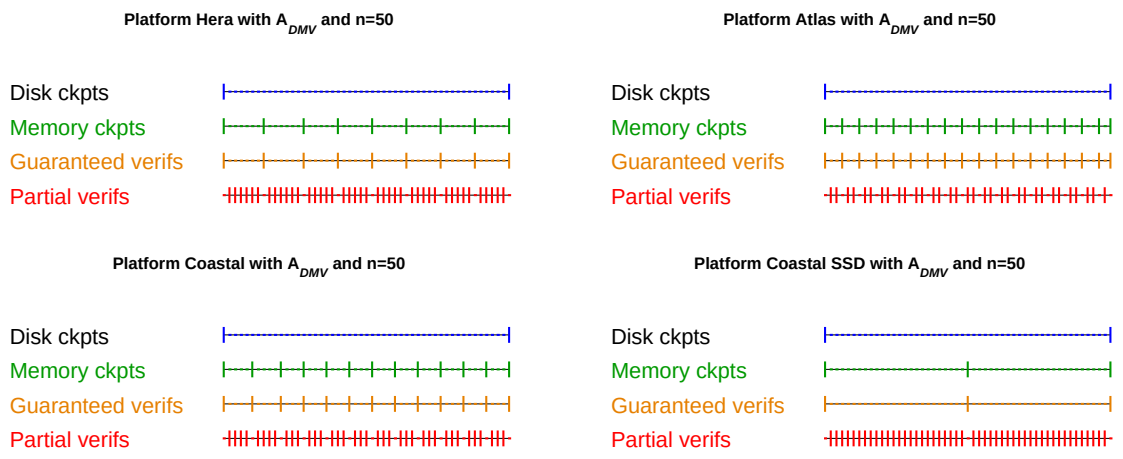


Figure 4.8: Distribution of disk checkpoints, memory checkpoints and verifications for the A_{DMV} algorithm on each platform with the *Uniform* pattern.

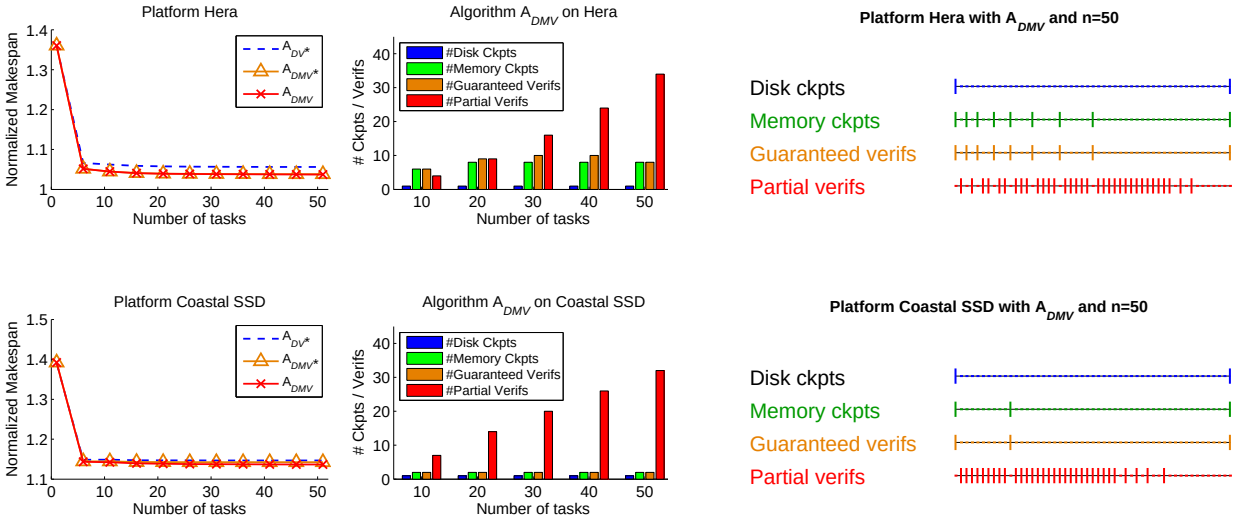


Figure 4.9: Performance of the three algorithms, and distribution of disk checkpoints, memory checkpoints and verifications (for the A_{DMV} algorithm) on platforms Hera and Coastal SSD with the *Decrease* pattern.

The second column shows the numbers of disk checkpoints, memory checkpoints, guaranteed and partial verifications given by the A_{DMV} algorithm. The third column is a visual representation of the corresponding solution obtained for 50 tasks and with the same configuration. We observe that the makespan obtained is very similar for all three algorithms (with a slight advantage for A_{DMV}). Since the large tasks at the beginning of the chain are more likely to fail, they will be checkpointed more often, as opposed to the small tasks at the end, which the algorithm does not even consider worth verifying.

***HighLow* pattern.** Once again, we focus on platforms Hera and Coastal SSD. Similarly to Figure 4.9, Figure 4.10 assesses the impact of the *HighLow* pattern on the performance of the

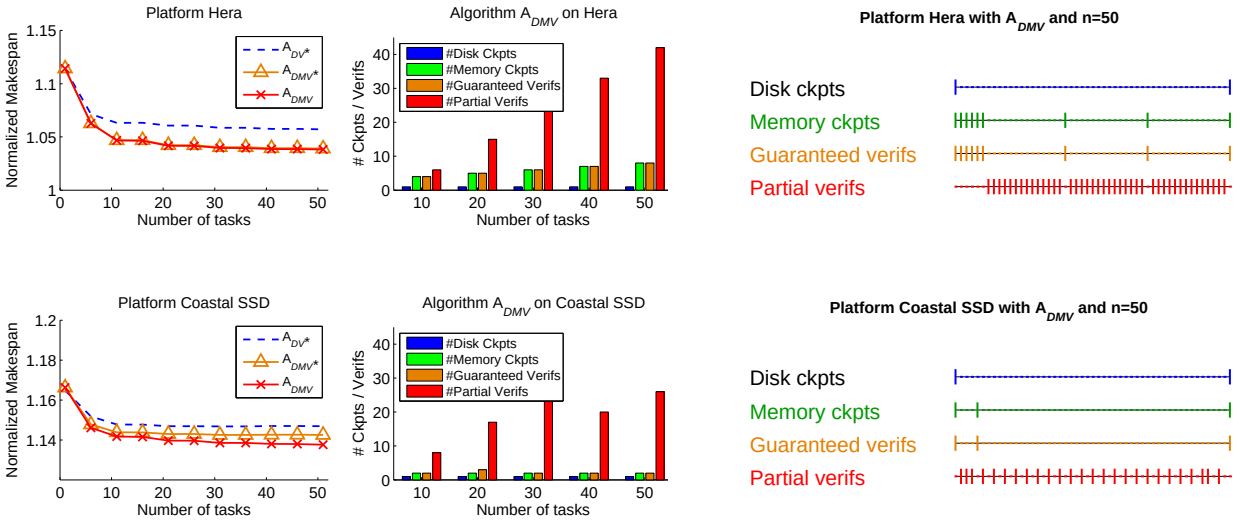


Figure 4.10: Performance of the three algorithms, and distribution of disk checkpoints, memory checkpoints and verifications (for the A_{DMV} algorithm) on platforms Hera and Coastal SSD with the *HighLow* pattern.

three algorithms as well as on the numbers and the positions of checkpoints and verifications. Recall that we set the first 10% of the tasks to contain 60% of the total computational weight, while the rest of the tasks contain the remaining 40%. With 50 tasks and a total computational weight of $25000s$, the first 5 tasks have a weight of $3000s$ each, while the remaining tasks have a weight of around $222s$ each. Under this configuration, an error occurring during the execution of a large task would cost $T^{\text{lost}} \approx 1500s$ time loss on average for fail-stop errors (see Equation (4.8)) and $3000s$ for silent errors, plus an additional $3000s$ time loss for each preceding task that has not been checkpointed. With the MTBF on Hera, a large task will fail with probability 1.3%, as opposed to the probability of 0.096% for small tasks. As a result, the disk checkpoint, which takes $300s$, turns out to be still too expensive, but the memory checkpoint, which takes only $15.4s$ on Hera, becomes mandatory: on average an error will occur way before the total accumulated cost of our preventive memory checkpoints even adds up to the cost of one task. On Coastal SSD, however, the memory checkpoint is still quite expensive, so that only one of the first 5 tasks is marked for verification and memory checkpointing. On both platforms, since the rest of the tasks are small, the solution is similar to the one we observed for the *Uniform* pattern, except that memory checkpoints and verifications are less frequent.

Summary of results. Overall, we observe that the combined use of disk checkpoints and memory checkpoints allows us to decrease the makespan, for the three task patterns and the four platforms. The use of partial verifications further decreases the makespan, especially on the Coastal SSD platform where the checkpointing costs are high. To give some numbers, our approach saves 2% of execution time on Hera and 5% on Atlas. These percentages may seem small, but they correspond to saving half an hour a day on Hera, and more than one hour a day on Atlas, with little further overhead.

Set	From	Level	3	2	1	Memory
(B)	Balaprakash et al. [6]	C (s)	150	50	30	10
		λ (Hz)	1.39e-6	6.94e-6	1.39e-5	2.78e-5

Table II
SET OF PARAMETERS (B) USED AS INPUT FOR SIMULATIONS.

4.5.2 Results for multi-level checkpointing

In this section, we perform additional experiments using a set of parameters that features $k = 3$ levels of disk checkpoints, as opposed to only one in the previous section. Therefore, we now focus on evaluating the impact of using multiple checkpointing levels to deal with fail-stop errors.

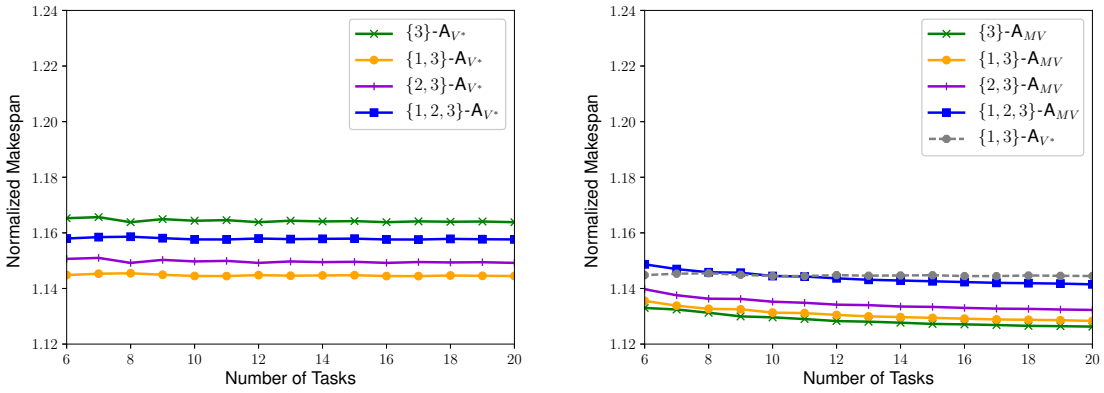
We compare three algorithms: (i) a multi-level algorithm A_{V^*} with up to $k = 3$ levels of disk checkpoints to handle both fail-stop and silent errors (with additional guaranteed verifications); and (ii) the combined algorithm A_{MV} that also uses memory checkpoints and partial verifications. Note that A_{MV} is the algorithm described in Section 4.4, while A_{V^*} is a simplification of this most sophisticated algorithm.

Platform settings. Table II presents the checkpointing costs and the associated error rates for this set of parameters, which are obtained from real measurements on the BG/Q platform Mira running LAMMPS application at ANL by Balaprakash et al. [6]. Multi-level checkpointing was provided by the FTI library [12], which offers four checkpointing levels (three levels of disk and one level of memory): local checkpoint (memory), local checkpoint + partner-copy (level-1 disk), local checkpoint + Reed-Solomon coding (level-2 disk), and PFS-based checkpoint (level-3 disk). The error rate corresponds to a default failure rate commonly used for petascale HPC applications [12, 37, 74].

Note that, with multiple levels of disk checkpoints, there is no obligation to use all available levels. In this particular case with $k = 3$ levels, one may choose among four possible subsets of levels: $\{3\}, \{1, 3\}, \{2, 3\}$, and $\{1, 2, 3\}$. Of course, we still have to account for all error types, which means that we need to adjust the error rates from the level selection as follows:

- $\{3\}$: use $\{\lambda_3 \leftarrow \lambda_1 + \lambda_2 + \lambda_3\}$;
- $\{1, 3\}$: use $\{\lambda_1\}$ and $\{\lambda_3 \leftarrow \lambda_2 + \lambda_3\}$;
- $\{2, 3\}$: use $\{\lambda_2 \leftarrow \lambda_1 + \lambda_2\}$ and $\{\lambda_3\}$;
- $\{1, 2, 3\}$: use $\{\lambda_1\}$, $\{\lambda_2\}$ and $\{\lambda_3\}$.

Impact of checkpointing level selection. Figure 4.11 presents the normalized makespan with respect to the error-free execution time obtained using the A_{V^*} algorithm (a) and A_{MV} algorithm (b), with up to 20 tasks under the *Uniform* pattern with total work $W = 3600s$. First, we observe that different level selections yield different overheads, but overall, using more levels does not always improve performance. In particular, we can see that, for the simple algorithm A_{V^*} without additional memory checkpoints or partial verifications to deal with silent errors, the best solution is to use the $\{1, 3\}$ level selection, which achieves an overhead around 14.5%. In comparison, using only level-3 checkpoints yields an overhead around 16.5%, while using all levels $\{1, 2, 3\}$ yields an overhead just below 16%. When allowing additional memory checkpoints and partial verifications with the A_{MV} algorithm, the $\{1, 3\}$ level selection no longer



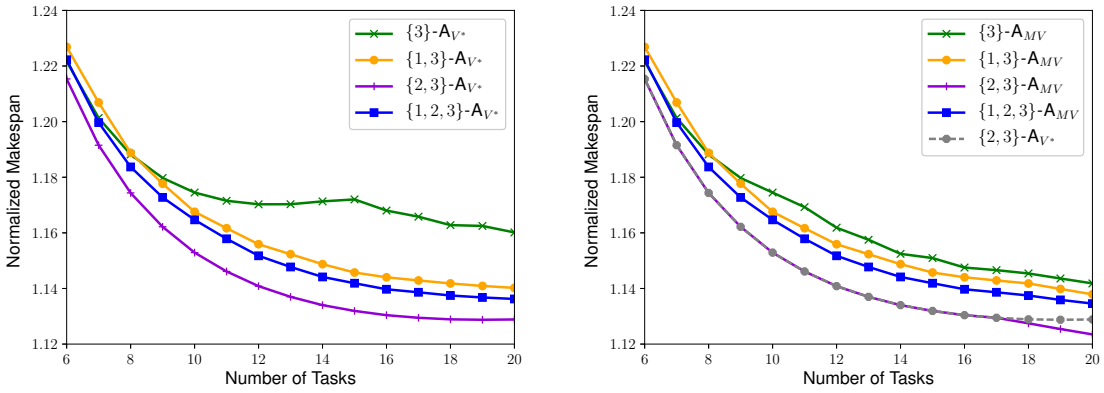
(a) (b)

Figure 4.11: Performance obtained by using the optimal solution to the MULTILEVEL-SILENT problem for settings (B), using the A_{V^*} algorithm (a) and the A_{MV} algorithm (b), under the *Uniform* pattern with total work $W = 3600s$.

achieves the best results. Instead, it appears that using only level-3 checkpoints, i.e., replacing level-1 checkpoints by the cheaper memory checkpoints, yields a slightly better overhead around 13%. Overall, A_{MV} improves upon the A_{V^*} algorithm (with only level-3 checkpoints and guaranteed verifications) under the best level selection $\{1, 3\}$ by 1.5%.

Similarly to Figure 4.11, Figure 4.12 presents the normalized makespan with respect to the error-free execution time obtained using the A_{V^*} algorithm (a) and A_{MV} algorithm (b), with up to 20 tasks under the *Uniform* pattern with total work $W = 25000s$. Note that tasks are now significantly larger than before, and we observe that the level selection $\{2, 3\}$ beats all the other possible combinations by achieving an overhead of 13% with the A_{V^*} algorithm (Figure 4.12a), which is further improved by another 0.5% when using the A_{MV} algorithm with additional memory checkpoints for 20 tasks (Figure 4.12b). Since larger tasks require *more* checkpoints, but offer *limited* opportunities to achieve that goal, the algorithms tend to favor additional levels of verified checkpoints, instead of single memory checkpoints, which will be lost when a fail-stop error strikes, or single verifications (either guaranteed or partial). This is why the A_{MV} algorithm becomes only slightly better with 20 tasks, and that it is not as helpful in this context.

Results for other patterns. Results for the *Decrease* and *HighLow* patterns for the combined problem are presented in Figures 4.13 and 4.14, respectively. As in the previous cases, we succeed to improve performance by combining the use of multi-level checkpointing and memory checkpoints (with verifications) for silent errors, especially when tasks are small (with total work $W = 3600s$). Note that Figure 4.14 shows a drastic drop in overhead between 10 tasks and 11 tasks. Indeed, with 10 or fewer tasks, the *HighLow* distribution consists of one big task, which has size $2160s$ when $W = 3600s$ and $15000s$ when $W = 25000s$. As a result, the probability of encountering an error (either fail-stop or silent) during the execution of the first task reaches 0.1 when $W = 3600s$ and 0.5 when $W = 25000s$, suggesting that task size plays an important role in the overhead. In comparison, with 11 or more tasks, and according to the



(a) (b)

Figure 4.12: Performance obtained by using the optimal solution to the MULTILEVEL-SILENT problem for settings (B), using the A_{V^*} algorithm (a) and the A_{MV} algorithm (b), under the *Uniform* pattern with total work $W = 25000s$.

HighLow distribution, we now have two big tasks instead of one. Therefore, the probability decreases to 0.05 when $W = 3600s$ and just under $1/3$ when $W = 25000s$.

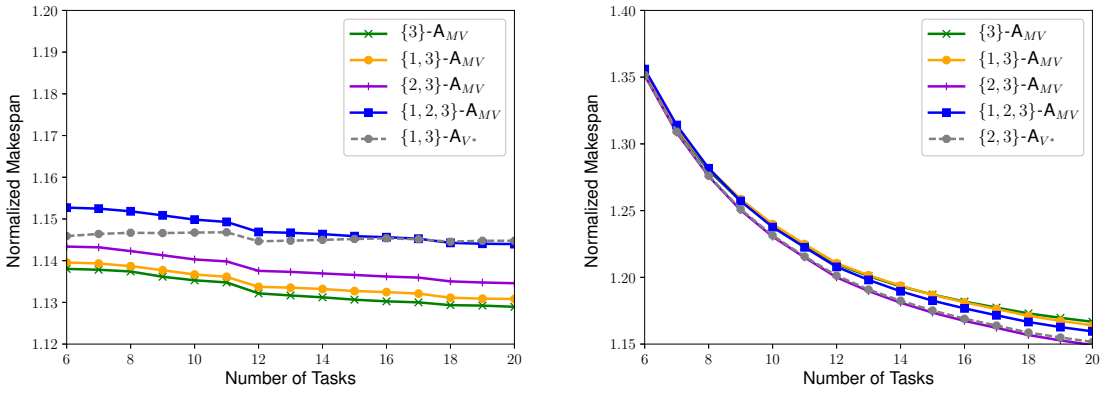
Summary of results. Overall, the simulation results have shown that the combined approach described in Section 4.4 to deal with both silent errors and multi-level fail-stop errors indeed leads to improved performance. In particular, when tasks are small enough, both approaches help equally to reduce the overhead. However, with fewer tasks and hence less freedom to checkpoint and verify, additional checkpoint levels seem to be favored over additional memory checkpoints or verifications. Furthermore, we have shown that the best checkpoint level selection does not always include all the levels. Finally, we remark that the implemented dynamic programming algorithms typically execute within just a few seconds and occupy up to 15GB of RAM for 20 tasks.

4.6 Related work

In this section, we discuss related work on fail-stop errors and silent errors, and finally outline specific results for linear workflows.

4.6.1 Fail-stop errors

The de-facto general-purpose error recovery technique in high performance computing is checkpoint and rollback recovery [28, 46]. For a divisible load application where checkpoints can be inserted at any point in execution for a nominal cost C, there exist well-known formulas due to Young [97] and Daly [36] to determine the optimal checkpointing period. For an application composed of a linear chain of tasks, as in this work, the problem of finding the optimal



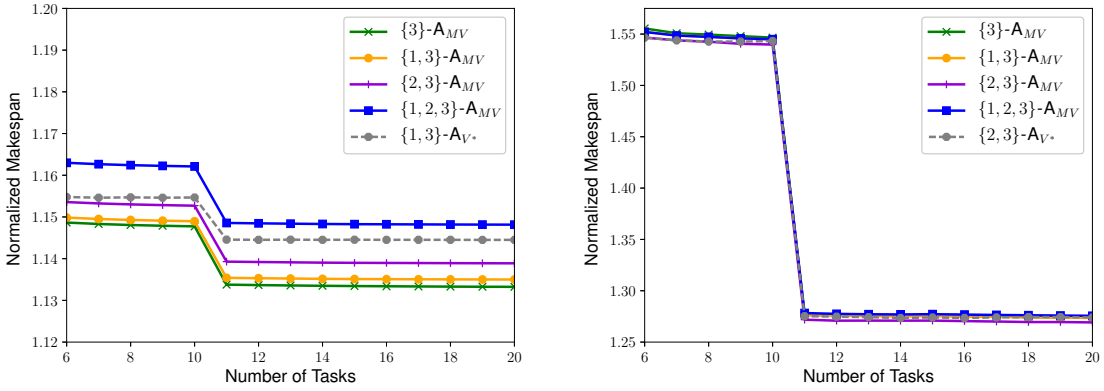
(a) (b)

Figure 4.13: Performance obtained by using the optimal solution to the MULTILEVEL-SILENT problem for settings (B), using the A_{V^*} algorithm under the *Decrease* pattern with total work $W = 3600s$ (a) and $W = 25000s$ (b).

checkpointing strategy, i.e., of determining which tasks to checkpoint, in order to minimize the expected execution time, has been solved by Toueg and Babaoglu [93].

However, single-level checkpointing schemes suffer from the intrinsic limitation that the cost of checkpointing/recovery grows with failure probability, and becomes unsustainable at large scale [17, 52] (even with diskless or incremental checkpointing [78]). To reduce the I/O overhead, various two-level checkpointing protocols have been studied. Vaidya [94] proposed a two-level recovery scheme that tolerates a single node failure using a local checkpoint stored on a partner node. If more than one failure occurs during any local checkpointing interval, the scheme resorts to the global checkpoint. Silva and Silva [88] advocated a similar scheme by using memory to store local checkpoints, which is protected by XOR encoding. Di et al. [39] analyzed a two-level computational pattern, and proved equal-length segments in the optimal solution. They also provided mathematical equations that can be solved numerically to compute the optimal pattern length and number of segments. In Chapter 2, we relied on disk checkpoints to cope with fail-stop failures and used memory checkpoints coupled with error detectors to handle silent data corruptions. We have derived first-order approximation formulas for the optimal pattern length as well as the number of memory checkpoints between two disk checkpoints.

Some authors have also generalized two-level checkpointing to account for an arbitrary number of levels. Moody et al. [74] implemented this approach in a three-level Scalable Checkpoint/Restart (SCR) library. They relied on a rather complex Markov model to recursively compute the efficiency of the scheme. Bautista-Gomez et al. [12] designed a four-level checkpointing library, called Fault Tolerance Interface (FTI), in which partner-copy and Reed-Solomon encoding are employed as two intermediate levels between local and global disks. Based on FTI, Di et al. [37] proposed an iterative method to compute the optimal checkpointing interval for each level with prior knowledge of the application's total execution time. We have provided a complete characterization of multi-level checkpointing pattern based on first-



(a) (b)

Figure 4.14: Performance obtained by using the optimal solution to the MULTILEVEL-SILENT problem for settings (B), using the A_{V^*} algorithm under the *HighLow* pattern with total work $W = 3600s$ (a) and $W = 25000s$ (b).

order approximation, thus generalizing Young/Daly's classical results in Chapter 3. Hakkarinen and Chen [60] considered multi-level diskless checkpointing for tolerating simultaneous failures of multiple processors. Balaprakash et al. [6] studied the trade-off between performance and energy for general multi-level checkpointing schemes.

4.6.2 Silent errors

Most traditional approaches maintain a single checkpoint. If the checkpoint file includes errors, the application faces an irrecoverable failure and must restart from scratch. This is because error detection latency is ignored in traditional rollback and recovery schemes, which assume instantaneous error detection (therefore mainly targeting fail-stop failures) and are unable to accommodate silent errors. We focus in this section on related work about silent errors. A comprehensive list of techniques and references is provided by Lu, Zheng and Chien [70].

Considerable efforts have been directed at error-checking to reveal silent errors. Error detection is usually very costly. Hardware mechanisms, such as ECC memory, can detect and even correct a fraction of errors, but in practice they are complemented with software techniques. The simplest technique is triple modular redundancy and voting [72], which induces a highly costly verification. For high-performance scientific applications, process replication (each process is equipped with a replica, and messages are quadruplicated) is proposed in the RedMPI library [53]. Elliot et al. [45] combine partial redundancy and checkpointing, and confirm the benefit of dual and triple redundancy. The drawback is that twice the number of processing resources is required (for dual redundancy). An approach based on checkpointing and replication is proposed in [75], in order to detect and enable fast recovery of applications from both silent errors and hard errors.

Application-specific information can be very useful to enable ad-hoc solutions, which dramatically decrease the cost of detection. Many techniques have been advocated. They include

memory scrubbing [64] and ABFT techniques [16, 63, 87], such as coding for the sparse-matrix vector multiplication kernel [87], and coupling a higher-order with a lower-order scheme for PDEs [14]. These methods can only detect an error but do not correct it. Self-stabilizing corrections after error detection in the conjugate gradient method are investigated by Sao and Vuduc [85]. Heroux and Hoemmen [19] design a fault-tolerant GMRES capable of converging despite silent errors. Bronevetsky and de Supinski [21] provide a comparative study of detection costs for iterative methods.

Recently, detectors based on data analytics have been proposed to serve as partial verifications [8, 10, 15]. These detectors use interpolation techniques, such as time series prediction and spatial multivariate interpolation, on scientific dataset to offer large error coverage for a negligible overhead. Although not perfect, their accuracy-to-cost ratios tend to be very high, which makes them interesting alternatives at large scale. For divisible load applications, periodic patterns with partial and guaranteed verifications are studied in Chapter 2. We point out that the approach described in this chapter is agnostic of the underlying error-detection technique and takes the cost of verification as an input parameter to the model.

4.6.3 Linear workflows

In this section, we focus on work related to linear workflows. The main difference with divisible load applications is that one can insert resilience mechanisms only at the end of the execution of a task. We may well have a limited number of tasks, which prevents the use of any periodic strategy à la Young/Daly [36, 97]. Instead, the optimal solution for any linear task graph is typically obtained with dynamic programming algorithms.

As already mentioned, Toueg and Babaoglu [93] have also dealt with single-level checkpointing for fail-stop errors. In a previous work [J3], their work has been extended to deal with silent errors in addition to fail-stop errors. The approach in [J3] uses only guaranteed verifications and one-level of checkpoint. It has been further extended in [W5] (the preliminary version of this work) to include partial verifications in addition to guaranteed verification, and in-memory checkpointing in addition to disk checkpointing.

This work provides the last step and shows how to add multi-level disk checkpointing protocols. We now deal with k disk checkpoint levels (where k is arbitrary), one memory checkpoint level, and partial and guaranteed verifications. As a result, we combine the most efficient techniques for fail-stop and silent errors within a unified framework.

4.7 Conclusion

In this chapter, we focused on HPC applications whose dependency graph forms a linear chain, and we proposed two important extensions to single-level checkpointing, allowing us to cope with both multi-level fail-stop errors and silent data corruptions, on large-scale platforms. Although numerous studies have dealt with either error source, few studies have dealt with both, while it is mandatory to address both sources simultaneously at scale. We have combined the multi-level disk checkpointing technique with in-memory checkpoints and verification mechanisms (partial or guaranteed), and we have designed a sophisticated multi-level dynamic pro-

gramming algorithm that computes the optimal solution for a linear application workflow in polynomial time.

Simulations based on realistic parameters on several platforms show consistent results, and confirm the benefit of the combined approach. Improvement can be seen both by using additional guaranteed and/or partial verifications for silent errors, and by selecting several levels of checkpoints, between those offered by the platform, to handle different types of fail-stop errors. While the most general algorithm has a high complexity of $O(n^{k+5})$, where n is the number of tasks and k is the number of checkpointing levels, it executes within a few seconds for $n = 20$ tasks and $k = 3$ levels, and therefore can be readily used for real-life linear workflows whose sizes rarely exceed tens of tasks.

Chapter 5

Voltage Overscaling Algorithms for Energy-Efficient Workflow Computations

In this chapter, we discuss several scheduling algorithms to execute tasks with voltage overscaling. Given a frequency to execute the tasks, operating at a voltage below threshold leads to significant energy savings but also induces timing errors. A verification mechanism must be enforced to detect these errors. As opposed to fail-stop or silent errors, timing errors are deterministic (but unpredictable). For each task, the general strategy is to select a voltage for execution, to check the result, and to select a higher voltage for re-execution if a timing error has occurred, and so on until a correct result is obtained. Switching from one voltage to another incurs a given cost, so it might be efficient to try and execute several tasks at the current voltage before switching to another one. In a preliminary version of this work, we have proposed an optimal polynomial dynamic programming algorithm to solve this problem for a linear chain of tasks [W6]. Determining the optimal solution for independent tasks turns out to be unexpectedly difficult. However, we provide the optimal algorithm for a single task, the optimal algorithm when there are only two voltages, and the optimal level algorithm for a set of independent tasks, where a level algorithm is defined as an algorithm that executes all remaining tasks when switching to a given voltage. Furthermore, we show that the optimal level algorithm is in fact globally optimal (among all possible algorithms) when voltage switching costs are linear. Finally, we report a comprehensive set of simulations to assess the potential gain of voltage overscaling algorithms. This work has been published in the proceedings of the *Pacific Rim International Symposium on Dependable Computing* (PRDC) [C5].

5.1 Introduction

Energy minimization has become a critical concern in High Performance Computing (HPC). Many authors have suggested to use *Dynamic Voltage and Frequency Scaling (DVFS)* to reduce the energy consumption during the execution of an application. Reducing the frequency (or speed) at which each core is operated is the most frequently advocated approach, and great savings have been demonstrated for a variety of scientific applications [7, 29, 57, 95]. However, reducing the voltage for a given speed may lead to even greater savings, because the total

consumed power is proportional to the square of the voltage. On the contrary, the dynamic power is linearly proportional to the frequency, and the static power is independent of it.

Keeping the same frequency and reducing the voltage is a promising direction which we explore in this chapter. There is no free lunch, though. Given a frequency, there is always a voltage recommended by the manufacturer, below which it might be unsafe to operate the core. This voltage always includes some environmental margin to be on the safe side. Near-threshold computing is a technique that consists in reducing the voltage below the recommended value, down to a *threshold voltage* V_{TH} (also called *nominal voltage*) that is still considered safe. Overscaling algorithms suggest to further reduce the operating voltage, at the risk of producing *timing errors*. Because the voltage is set to a very low value, the results of some logic gates could be used before their output signals reach their final values, which could possibly lead to an incorrect result. The occurrence of a timing error depends upon many parameters: the voltage and frequency, and the nature of the target operation: different operations within the ALU may have different critical-path lengths. But in addition, for a given operation, different sets of operands may lead to different critical-path lengths (to see this, take a simple addition and think of a carry rippling to different gates depending upon the operands). Timing errors are therefore very different from usual fail-stop failures or silent errors that are dealt with in the literature: they are not random but, instead, they are purely deterministic. Indeed, replaying the same operation with the same set of operand under the same conditions will lead to the same result. Although deterministic, timing errors are unpredictable, because it is not possible to test all possible operands for a given operation. Therefore, for a given operation[1], an error probability is associated to each voltage and represents the fraction of operands for which incorrect results will be produced by executing that operation on these operands at that voltage.

We need to take actions to mitigate the timing errors striking when voltage is aggressively lowered. After executing a task, we insert a verification mechanism to check the correctness of the result. In our study, the scheduling algorithms are agnostic of the nature of this verification mechanism, which could be anything from (costly) duplication to (cheap) checksumming and other application-specific methods. Of course the cheaper the cost of the verification, the smaller the overhead to total execution time.

To execute a given task, scheduling algorithms with voltage overscaling operate as follows: they are given a (discrete) set of possible voltages to operate with, and one of them as an input voltage. The first decision to take is whether to execute the task at that voltage or to choose another one. In the latter case, there is a *switching cost* to pay to change voltage. Regardless of the decision, the result is verified after the execution of the task. If the verification mechanism returns that the result is correct, we are done. If not, we need to re-execute the task. Remember that timing errors are deterministic: there is no point in re-executing the task with the same voltage, we know that we will get the same error. We need to select a higher voltage that will reduce the probability of failure, paying a switching cost, and re-execute the task with this voltage. Because the voltage is higher, the error probability is reduced, and we have a chance that the second execution is correct. The higher the second voltage, the better that chance, but the higher the cost of the execution, so there is a trade-off to achieve. If we are unlucky, we may have to try several higher and higher voltages, up to eventually finishing

[1] and a given frequency: remember that throughout the text, we assume the frequency to be given.

by using the threshold voltage, which is 100% safe but very costly. In Section 5.4, we give the optimal scheduling algorithm for a single task, extending our previous result [W6] to the case where an input voltage is given to the algorithm.

The problem gets more complicated when there are many tasks to schedule. We assume that these tasks correspond to the same operation but involve different operands (think of a collection of matrix products or stencil updates). Given a voltage, each task has the same probability to fail. In the absence of switching costs (an unrealistic assumption in practice), the tasks can be dealt with independently. However, to amortize the switching cost from a given voltage to a new one, it might be a good idea to try and execute several tasks (or even all the remaining tasks) at a given voltage. One key contribution of this work is to analyze *level algorithms*, which always execute all the remaining tasks once a voltage has been selected. We provide a dynamic programming algorithm that computes the optimal level algorithm as a function of the input voltage costs and error probabilities, and of the number of tasks to execute.

Level algorithms turn out to be dominant among all possible algorithms when voltage switching costs are linear. Technically, if we have three voltages $V_1 < V_2 < V_3$, linear switching costs means that $s_{1,3} = s_{1,2} + s_{2,3}$, where $s_{i,j}$ is the cost to switch from V_i to V_j (or the other way round, from V_j to V_i). With linear switching costs, we show that the optimal level algorithm is in fact optimal among all possible algorithms, not just linear ones.

Finally, an important contribution of the chapter is to experimentally assess the usefulness of voltage overscaling algorithms. We first consider a case study from numerical linear algebra, where tasks are matrix-products that can be verified through ABFT checksums. We then envision different scenarios where we study the impact of each parameter (verification cost, voltage cost and error probability, switching costs). In addition to the gain in energy consumption, we also investigate the performance degradation: while we keep the same frequency (thereby avoiding a global slowdown of the execution as is the case with DVFS), we do have two sources of performance overhead: (i) the verification mechanism, and (ii) the time lost due to re-execution(s) and voltage switching after timing errors.

To the best of our knowledge, this work (together with our initial workshop paper [W6]) is the first algorithmic approach for voltage overscaling. Previous studies are hardware oriented and require special hardware mechanisms to detect timing errors [51, 66, 68, 80]. On the contrary, we propose scheduling algorithms that can be called by the operating system of the platform.

The rest of this chapter is organized as follows. In Section 5.2, we introduce a formal model for timing errors. Then in Section 5.3, we illustrate several scheduling strategies by working out a couple of toy examples. We present the optimal algorithm for a single task in Section 5.4, and move to scheduling several tasks in Section 5.5. We report the results of a comprehensive set of simulations to assess the impact and benefits of the previous voltage overscaling algorithms in Section 5.6. Finally, we provide concluding remarks in Section 5.7.

5.2 Model

We now present a formal model for timing errors. We introduce the main notations and assumptions, and investigate the impact of timing errors on the success and failure probabilities of

tasks. Because timing errors are deterministic, conditional properties are completely different from what is usually enforced for the resilience of HPC applications.

5.2.1 Timing errors

As already mentioned, we focus on a fixed frequency environment (this frequency may have been chosen to achieve a given performance). Then timing errors depend upon the voltage selected for execution, and we model this with the following two assumptions:

Assumption 1. *Given an operation and an input I (the set of operands), there exists a* threshold voltage $V_{\text{TH}}(I)$*: using any voltage V below the threshold ($V < V_{\text{TH}}(I)$) will always lead to an incorrect result, while using any voltage above that threshold ($V \geq V_{\text{TH}}(I)$) will always lead to a successful execution. Note that different inputs for the same operation may have different threshold voltages.*

Assumption 2. *When an operation is executed under a given voltage V, there is a probability p_V that the computation will fail, i.e., produces at least one error, on a random input. This failure probability is computed as $p_V = |\mathcal{I}_f(V)|/|\mathcal{I}|$, where $\mathcal{I}$ denotes the set of all possible inputs and $\mathcal{I}_f(V) \subseteq \mathcal{I}$ denotes the set of inputs for which the operation will fail at voltage V. Equivalently, $\mathcal{I}_f(V)$ is the set of inputs whose threshold voltages are strictly larger than V, according to Assumption 1. For any two voltages V_1 and V_2 with $V_1 \geq V_2$, we have $\mathcal{I}_f(V_1) \subseteq \mathcal{I}_f(V_2)$ (Assumption 1), hence $p_{V_1} \leq p_{V_2}$.*

If a task consists of t identical operations, each executed at voltage V with error probability p_V, then the probability of successful execution of the task is $(1 - p_V)^t$: the larger the task, the greater the risk. Since timing errors are essentially silent errors, they do not manifest themselves until the corrupted data has led to an unusual application behavior, which may be detected long after the error has occurred, wasting the entire computation done so far. Hence, an error-detection mechanism (also called *verification* mechanism) is necessary to ensure timely detection of timing errors after the execution of each task. In this work, we assume that this mechanism is given. All the algorithms presented in Sections 5.4 and 5.5 are fully general and agnostic of the error-detection technique (checksum, error correcting code, coherence tests, etc.).

5.2.2 Notations

We consider a set $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$ of n tasks to be executed by the system. All tasks share the same computational weight, including the work to verify the correctness of the result at the end. Hence, all tasks have the same execution time and energy consumption under a fixed voltage. We apply *Dynamic Voltage OverScaling (DVOS)* to tradeoff between energy cost and failure probability. The platform can choose an operating voltage among a set $\mathcal{V} = \{V_1, V_2, \cdots, V_k\}$ of k discrete values, where $V_1 < V_2 < \cdots < V_k$. Each voltage V_ℓ has an *energy cost* per task c_ℓ that increases with the voltage, i.e., $c_1 < c_2 < \cdots < c_k$. Based on Assumption 2, each voltage V_ℓ also has a *failure probability* p_ℓ that decreases with the voltage, i.e., $p_1 > p_2 > \cdots > p_k$. We assume that the highest voltage V_k equals the *nominal* voltage V_{TH}

with failure probability $p_k = 0$, thus guaranteeing error-free execution for all possible inputs. For convenience, we also use a *null* voltage V_0 with failure probability $p_0 = 1$ and null energy cost $c_0 = 0$. Here is a summary of these notations:

Voltages	V_0	V_1	V_2	$\cdots$	$V_k = V_{\text{TH}}$
Failure Prob.	$p_0 = 1$	p_1	p_2	$\ldots$	$p_k = 0$
Energy cost	$c_0 = 0$	c_1	c_2	$\ldots$	c_k

We further assume that DVOS is only applied to reduce energy consumption, and leaving the system at a voltage below V_{TH} after the execution of the task(s) is not allowed (because it would not be safe). This is formally stated in the following assumption.

Assumption 3. *The system is initially running at nominal voltage V_{TH} and its voltage must be reset back to V_{TH} when all computations are done.*

Switching the operating voltage also incurs an energy cost. Let $s_{\ell,h}$ denote the energy consumed to switch the system's operating voltage from V_ℓ to V_h. We have $s_{\ell,h} = 0$ if $\ell = h$ and $s_{\ell,h} > 0$ if $\ell \neq h$. Moreover, we make the following assumptions on the properties of the switching costs, which are true in many systems in practice:

Assumption 4. *The voltage switching costs of the system satisfy to:*
- Symmetry*: $s_{\ell,h} = s_{h,\ell}$ for all $V_\ell, V_h \in \mathcal{V}$;*
- Dominance*: $s_{\ell,h} \geq s_{\ell,p}$ and $s_{\ell,h} \geq s_{p,h}$ for $V_\ell \leq V_p \leq V_h$;*
- Triangle inequality*: $s_{\ell,h} \leq s_{\ell,p} + s_{p,h}$ for $V_\ell \leq V_p \leq V_h$.*

Definition 4 (Linear switching costs)**.** *We have linear costs when the triangle inequality is always an equality: $s_{\ell,h} = s_{\ell,p} + s_{p,h}$ for all $V_\ell \leq V_p \leq V_h$.*

The objective is to minimize the expected total energy consumption by determining the optimal strategy to execute the set of tasks.

5.2.3 Success and failure probabilities

We now consider the implications of Assumptions 1 and 2 on the success and failure probabilities of executing a task following a sequence of voltages. For the ease of writing, we assume that the execution of each task has already failed under the null voltage V_0 (at energy cost $c_0 = 0$).

Lemma 10. *Consider a sequence $\langle V_1, V_2, \cdots, V_m \rangle$ of m voltages, where $V_1 < V_2 < \cdots < V_m$, under which a given task is executed. For any voltage V_ℓ, where $1 \leq \ell \leq m$, given that the execution of the task on a certain input has already failed under voltages $V_0, V_1, \cdots, V_{\ell-1}$, the probability that the task execution will fail or succeed under voltage V_ℓ on the same input is*

$$\begin{aligned} \mathbb{P}(V_\ell\text{-fail} \mid V_0 V_1 \cdots V_{\ell-1}\text{-fail}) &= \frac{p_\ell}{p_{\ell-1}}, \\ \mathbb{P}(V_\ell\text{-succ} \mid V_0 V_1 \cdots V_{\ell-1}\text{-fail}) &= 1 - \frac{p_\ell}{p_{\ell-1}}. \end{aligned}$$

Proof. We prove the probabilities using Assumptions 1 and 2. The task under study is the execution of some computation on some input I. Since this task execution has failed under voltages $V_0, V_1, \cdots, V_{\ell-1}$, we know that input I satisfies $I \in \bigcap_{h=0}^{\ell-1} \mathcal{I}_f(V_h) = \mathcal{I}_f(V_{\ell-1})$ ($\mathcal{I}_f(V_h)$ is the set of inputs on which the computation fails under voltage V_h). Then, the task execution will fail again under voltage V_ℓ if the input satisfies $I \in \mathcal{I}_f(V_\ell) \subseteq \mathcal{I}_f(V_{\ell-1})$. Otherwise, the task execution will succeed. Given that the input is randomly chosen (we have no a priori knowledge on it), the failure probability is

$$\mathbb{P}(V_\ell\text{-fail} \mid V_0 V_1 \cdots V_{\ell-1}\text{-fail}) = \frac{|\mathcal{I}_f(V_\ell)|}{|\mathcal{I}_f(V_{\ell-1})|} = \frac{|\mathcal{I}_f(V_\ell)|/|\mathcal{I}|}{|\mathcal{I}_f(V_{\ell-1})|/|\mathcal{I}|} = \frac{p_\ell}{p_{\ell-1}},$$

and the success probability is

$$\mathbb{P}(V_\ell\text{-succ} \mid V_0 V_1 \cdots V_{\ell-1}\text{-fail}) = \frac{|\mathcal{I}_f(V_{\ell-1}) \setminus \mathcal{I}_f(V_\ell)|}{|\mathcal{I}_f(V_{\ell-1})|} = 1 - \frac{|\mathcal{I}_f(V_\ell)|}{|\mathcal{I}_f(V_{\ell-1})|} = 1 - \frac{p_\ell}{p_{\ell-1}}.$$

□

5.3 Examples

In this section, we focus on the special case where there are only two available voltages ($k = 2$). In the first study, for the sake of illustration, we assume that Assumption 3 does not hold and that the system is originally at voltage V_1. In the second study, we derive the optimal policy, for two voltages, assuming that Assumption 3 holds (which will always be the case in the following sections).

5.3.1 Two voltages without Assumption 3

The system is initially at voltage V_1. We assume that there are only two tasks to execute. It is twice as expensive energy-wise to execute a task at V_2 than at V_1 ($c_1 = 1$ and $c_2 = 2$), the failure probability at V_1 is 80% ($p_1 = 0.8$), and switching voltage from V_1 to V_2 is as expensive as executing 20 tasks at V_1 ($s_{1,2} = 20$). If there were no switching costs ($s_{1,2} = 0$), no task would be executed at voltage V_1. Indeed, executing a task at voltage V_2 costs $c_2 = 2$, when the expected cost of executing a task at V_1 is $c_1 + p_1 \cdot c_2 = 1 + 0.8 \cdot 2 = 2.6$. If we could switch voltages for free, we would always switch to the nominal voltage to execute tasks. However, the switching cost is expensive, here, and the system is initially at voltage V_1. We can envision three policies:

- Switch directly to V_2 to execute both tasks. This costs: $s_{1,2} + 2 \cdot c_2 = 24$.
- Execute each task at V_1. If all executions succeed (which happens with probability $(1 - p_1)^2$), we are done. Otherwise, switch to V_2 and re-execute the failed tasks. The expected cost is $2 \cdot c_1 + (1 - (1 - p_1)^2) s_{1,2} + 2 \cdot p_1 \cdot c_2 = 24.4$.

- Execute the first task at V_1 and switch to V_2 after its completion if and only if its execution failed. The rationale is as follows. If the execution of the first task fails, whatever the result of the execution of the second task we will have to switch to V_2 to re-execute the first task. Moreover, we have shown that executing a task at V_2 is cheaper than trying to execute it at V_1 and then re-execute it at V_2 in case of failure. Therefore, we save energy by not attempting to execute the second task at V_1. If the execution of the first task was successful, we remain at V_1 in (the unlikely) case that the execution of the second case will also succeed and that we will never have to switch to V_2 (thus saving the huge switching cost). The expected cost of this policy is then: $c_1 + p_1(s_{1,2} + 2 \cdot c_2) + (1 - p_1)(c_1 + p_1(s_{1,2} + c_2)) = 23.92$.

The third policy is optimal in this case. This example illustrates two important facts: (i) the optimal policy may be dynamic, the decision at which voltage the next task should be executed depending on the success or failure of *other* tasks; (ii) switching costs can have a significant impact on the shape of the optimal solution.

5.3.2 Two voltages under Assumption 3

We now assume that we have n tasks to process, and that Assumption 3 holds (as it should). We have two cases to consider, whether V_1 is used or not.

- V_1 is not used. The cost is then $n \cdot c_2$.
- V_1 is used. Then, whatever the policy, the two switching costs $s_{1,2}$ and $s_{2,1}$ will be payed. Executing a task at voltage V_2 costs c_2. Executing a task first at voltage V_1 costs: $c_1 + p_1 \cdot c_2$. Therefore, it may only be worth using V_1 if $c_1 + p_1 \cdot c_2 \leq c_2 \Leftrightarrow c_1 \leq (1 - p_1)c_2$. In this case, an optimal solution is to first execute *all* tasks at V_1, then the voltage must be switched back to V_2 and, finally, failed tasks must be re-executed. Indeed, whatever happens, both switching costs are paid, the solution is the same as if switching costs were zero The expected cost is then: $2s_{1,2} + n \cdot c_1 + n \cdot p_1 \cdot c_2$. This is better than the first policy if and only if:

$$n \cdot c_2 \geq 2s_{1,2} + n \cdot c_1 + n \cdot p_1 \cdot c_2 \Leftrightarrow n \geq \left\lceil \frac{2s_{1,2}}{c_2(1 - p_1) - c_1} \right\rceil$$

The optimal strategy is thus: if $c_1 \leq (1 - p_1)c_2$ and $n \geq \lceil \frac{2s_{1,2}}{c_2(1-p_1)-c_1} \rceil$, then switch to V_1, execute all tasks at V_1, switch to V_2 and re-execute all failed tasks; otherwise, stay at V_2 and execute all tasks at that voltage. Hence, the shape of the optimal strategy depends of the number of tasks to execute. Note that thanks to Assumption 3, the shape of the solution is simpler than in the previous study.

5.4 Scheduling for a single task

We focus here on the special case where there is only one task to execute. We present an optimal algorithm for this case, extending our previous result [W6] to the case where an input voltage is given to the algorithm. We need this extension to prepare for the general case with several tasks: indeed, consider a simple algorithm that proceeds task after task. For the first

task, the input voltage is always V_{TH}, according to Assumption 3. But when the execution of the first task completes, the input voltage for the second task may well be different from V_{TH}, if for instance the algorithm tried, and succeeded with, a lower voltage.

We first define some notations. Let $E^*_{\text{ONE}}(V_h, V_\ell)$ denote the optimal expected energy needed to execute the task starting from voltage V_h, given that the task has previously failed under a sequence of lower voltages, the highest one among which is V_ℓ. Let $E^*_{\text{ONE}}(V_\ell)$ and $V^*_{\text{ONE}}(V_\ell)$ denote, respectively, the optimal expected energy and the optimal voltage needed to execute the task after it has just failed under voltage V_ℓ. The following theorem shows the optimal solution:

Theorem 17. *To execute a single task on a system with k voltages, the optimal expected energy consumption as well as the optimal sequence of voltages can be obtained by dynamic programming with complexity $O(k^2)$.*

Proof. Suppose the task has just failed under voltage V_ℓ and it is about to be executed at a higher voltage $V_h > V_\ell$. According to Lemma 10, the probability that the tasks will fail again under V_h is given by $\mathbb{P}(V_h\text{-fail} \mid V_\ell\text{-fail}) = p_h/p_\ell$. If the task is successfully completed at V_h, we can just reset the voltage from V_h back to the nominal voltage V_k. Otherwise, we need to determine the optimal voltage $V^*_{\text{ONE}}(V_h)$ to continue executing the task until successful completion. Thus, for any pair (V_h, V_ℓ) of voltages with $V_0 \le V_\ell < V_h \le V_k$, we can compute $Eone^*(V_h, V_\ell)$ by the following dynamic programming formulation:

$$E^*_{\text{ONE}}(V_h, V_\ell) = c_h + \left(1 - \frac{p_h}{p_\ell}\right) s_{h,k} + \frac{p_h}{p_\ell} \cdot \min_{h<p\le k} \left\{ s_{h,p} + E^*_{\text{ONE}}(V_p, V_h) \right\}. \quad (5.1)$$

The table is initialized with $E^*_{\text{ONE}}(V_k, V_\ell) = c_k$ regardless of V_ℓ, and the entire table can be computed in $O(k^2)$ time.

The optimal expected energy to execute the task after it has just failed under voltage V_ℓ is therefore

$$E^*_{\text{ONE}}(V_\ell) = \min_{\ell<h\le k} \left\{ s_{\ell,h} + E^*_{\text{ONE}}(V_h, V_\ell) \right\}, \quad (5.2)$$

and the optimal voltage to execute the task in this case is $V^*_{\text{ONE}}(V_\ell) = V_{h'}$, where $h' = \arg\min_{\ell<h\le k} \left\{ s_{\ell,h} + E^*_{\text{ONE}}(V_h, V_\ell) \right\}$. Again, computing $E^*_{\text{ONE}}(V_\ell)$ and $V^*_{\text{ONE}}(V_\ell)$ for all $1 \le \ell \le k$ takes $O(k^2)$ time. The optimal expected energy to execute the task from the initial nominal voltage V_k is thus:

$$E^*_{\text{ONE}} = \min_{1\le h\le k} \left\{ s_{k,h} + E^*_{\text{ONE}}(V_h, V_0) \right\},$$

where V_0 is the null voltage with failure probability $p_0 = 1$. The optimal starting voltage to execute the task is $V^*_{\text{ONE}} = V_{h'}$, where $h' = \arg\min_{1\le h\le k} \left\{ s_{k,h} + E^*_{\text{ONE}}(V_h, V_0) \right\}$. This can be computed in $O(k)$ time. □

5.5 Scheduling for several tasks

We now consider the general case of executing a set of n independent tasks, where $n \ge 2$. All tasks correspond to the same computational operations, but may have different threshold

voltages, because they operate on different data sets. The problem turns out to be more complicated than expected. We start, in Section 5.5.1, with the introduction of two simple scheduling strategies, *task-by-task* and *level*, before sketching the description of a general scheduling algorithm. Then we show how to determine the optimal level algorithm in Section 5.5.2. Finally, in Section 5.5.3, we prove that the optimal level algorithm is in fact globally optimal among all possible scheduling algorithms when switching costs are linear.

5.5.1 Scheduling algorithms and strategies

We now provide an informal description of scheduling algorithms for independent tasks. There are two simple execution strategies, which we call *task-by-task* and *level*. The task-by-task strategy considers the tasks one after the other, waiting for the successful completion of the current task before proceeding to the (first) execution of the next task. The task-by-task strategy relies upon the optimal algorithm for a single task and input voltage described in Section 5.4. After the successful execution of the current task, the platform voltage is set at some value V_h, which we use as input voltage for applying the optimal single-task algorithm to the execution of the next task.

While optimal for each task, this strategy may end up paying many switching costs. For instance, if the optimal single-task algorithm always starts with some low voltage, say V_s, regardless of the input voltage V_h, then the task-by-task strategy will have to switch back down to V_s each time there has been a timing error in the execution of the previous task.

To minimize switching costs, when given an input voltage, another strategy is to execute all tasks at that voltage, before switching to another voltage and execute all remaining tasks at that other voltage, and so on. This *level* strategy goes voltage-by-voltage instead of task-by-task, executing all tasks at a given voltage before switching to another one (hence its name).

While very natural, the task-by-task and level strategies are not the only possible algorithms. In fact, a general scheduling algorithm proceeds as follows. At each step, we are given an input voltage (that of the last execution of a task, or V_{TH} initially) and the list of remaining tasks, together with their history (the last voltage tried for execution of each task is recorded, with V_0 for initial condition). Then the algorithm selects one task in the list and one voltage V_{new} higher than the one recorded for this task[2], and executes the task at that voltage. A switching cost is paid if V_{new} is different from the input voltage. If the execution is successful, the task is removed from the list, and otherwise, the task stays in the list, and its history is updated with V_{new} being recorded. The algorithm then proceeds to the next step, with V_{new} as input voltage. The key decision at each step of the scheduling algorithm is the selection of the new task and voltage pair, and this decision may well depend upon the number and history of the tasks in the list, the input voltage, and all the problem parameters (cost and error probability of each voltage, and switching costs). Altogether, we have a complex decision to make at each step, and it seems very difficult to prove the optimality of a scheduling algorithm in the general case.

[2]There would be no point in trying the recorded voltage again, or a lower one: we know that the execution of the task would fail again.

5.5.2 Level algorithms

In this section, we formally define level algorithms, and we provide a dynamic programming algorithm to compute the optimal level algorithm.

Definition 5 (Level algorithms)**.** *A level algorithm executes a set of independent tasks as follows:*

1. *Select the initial voltage V;*
2. *Switch to voltage V and execute all remaining tasks;*
3. *Remove the successfully completed tasks from the set;*
4. *If there are still some tasks not successfully completed chose the next, higher, voltage V to try and go to Step 2;*
5. *If the last voltage used is not V_k, switch to voltage V_k.*

We call the sequence of voltages tried by a level algorithm the voltage sequence.

The level strategy guarantees that each voltage will be used at most once, so the voltages will not change more than k times during the execution of the entire set. We have to determine the optimal sequence of voltages to characterize the optimal level algorithm. Before presenting the dynamic programming algorithm that solves this problem, we need a few notations.

Let $E^*_{\text{SET}}(i, V_h, V_\ell)$ denote the optimal expected energy needed to execute i tasks starting at voltage V_h, provided that these tasks have just failed under a lower voltage $V_\ell < V_h$. Then, the optimal expected energy to execute i tasks that have just failed under voltage V_ℓ is given by $E^*_{\text{SET}}(i, V_\ell) = \min_{\ell<h\leq k} \left\{ s_{\ell,h} + E^*_{\text{SET}}(i, V_h, V_\ell) \right\}$.

Theorem 18. *To execute a set of n independent tasks on a system with k voltages, the optimal expected energy consumption as well as the next optimal voltage to execute a given number of tasks —which failed at a lower voltage or knowing that we are at nominal voltage and before any execution— can be obtained by dynamic programming with complexity $O(n^2k^2)$.*

Proof. Suppose a set of i tasks that have just failed under voltage V_ℓ and that will be executed at a higher voltage V_h. According to Lemma 10, the probability that any of these tasks fails again at V_h is $\mathbb{P}(V_h\text{-fail} \mid V_\ell\text{-fail}) = p_h/p_\ell$. Thus, the probability that j tasks will remain uncompleted after they are all executed at voltage V_ℓ is

$$\mathbb{P}(j \text{ tasks remain}) = \binom{i}{j} \left(\frac{p_h}{p_\ell}\right)^j \left(1 - \frac{p_h}{p_\ell}\right)^{i-j}$$

for any $0 \leq j \leq i$. If no task remains, e.g., $j = 0$, we need to reset the voltage from V_h back to the nominal voltage V_k. Otherwise, we need to determine the optimal voltage to execute the

remaining j tasks. Hence, the dynamic program:

$$\begin{aligned}
&E^*_{\text{SET}}(i, V_h, V_\ell) = i \cdot c_h + \mathbb{P}(\text{no task remains}) \cdot s_{h,k} \\
&+ \sum_{j=1}^{i} \left(\mathbb{P}(j \text{ tasks remain}) \min_{h<p\le k} \{ s_{h,p} + E^*_{\text{SET}}(j, V_p, V_h) \} \right) \\
&= i \cdot c_h + \left(1 - \frac{p_h}{p_\ell}\right)^i s_{h,k} \\
&+ \sum_{j=1}^{i} \left(\binom{i}{j} \left(\frac{p_h}{p_\ell}\right)^j \left(1 - \frac{p_h}{p_\ell}\right)^{i-j} \cdot \min_{h<p\le k} \{ s_{h,p} + E^*_{\text{SET}}(j, V_p, V_h) \} \right)
\end{aligned}$$

for all $1 \le i \le n$ and all (V_h, V_ℓ) pairs with $V_0 \le V_\ell < V_h \le V_k$. In particular, when $V_h = V_k$, we have $E^*(i, V_k, V_\ell) = i \cdot c_k$ for all $1 \le i \le n$ regardless of V_ℓ.

The optimal expected energy needed to execute i remaining tasks after they have just failed under voltage V_ℓ is therefore

$$E^*_{\text{SET}}(i, V_\ell) = \min_{\ell<h\le k} \left\{ s_{\ell,h} + E^*_{\text{SET}}(i, V_h, V_\ell) \right\}. \tag{5.3}$$

The optimal voltage to execute these tasks is $V^*_{\text{SET}}(i, V_\ell) = V_{h'}$, where $h' = \arg\min_{\ell<h\le k} \left\{ s_{\ell,h} + E^*_{\text{SET}}(i, V_h, V_\ell) \right\}$. The optimal expected energy needed to execute all n tasks, given that the initial system voltage is the nominal voltage V_k, is

$$E^*_{\text{SET}} = \min_{1\le h\le k} \left\{ s_{k,h} + E^*_{\text{SET}}(n, V_h, V_0) \right\},$$

where V_0 is the null voltage with failure probability $p_0 = 1$. The optimal starting voltage to execute the entire set is $V^*_{\text{SET}} = V_{h'}$, where $h' = \arg\min_{1\le h\le k} \left\{ s_{k,h} + E^*_{\text{SET}}(n, V_h, V_0) \right\}$.

The complexity is clearly dominated by the computation of $E^*_{\text{SET}}(i, V_s, V_\ell)$ for all $1 \le i \le n$, $V_0 \le V_\ell < V_k$ and $V_\ell < V_s \le V_k$, which takes $O(n^2k^2)$ time. □

Theorem 18 shows that the optimal voltage to select after each iteration depends on the number of remaining tasks. To demonstrate this point, consider an example with three voltages and 10 independent tasks. The energy costs of the voltages are $c_1 = 0.1$, $c_2 = 1$ and $c_3 = 5$, and the corresponding error probabilities are $p_1 = 0.8$, $p_2 = 0.5$ and $p_3 = 1$. The voltage switching costs are $s_{1,2} = s_{2,3} = 1$ and $s_{1,3} = 1.1$. According to Theorem 18, the optimal voltage to start executing the tasks is V_1. Suppose V_1 has been used. We consider the following two cases.

- Case 1: There is only 1 task left. In this case, switching first to V_2 and in case of failure again to V_3 incurs an expected cost of $s_{1,2} + c_2 + \frac{p_2}{p_1}(s_{2,3} + c_3) + \left(1 - \frac{p_2}{p_1}\right) s_{2,3} = 6.125$. On the other hand, switching directly to V_3 incurs a total cost of $s_{1,3} + c_3 = 6.1$. Hence, the best strategy in this case is to switch directly to V_3.

- Case 2: There are 9 tasks left. Then, switching first to V_2 and then to V_3 incurs an expected cost of $s_{1,2} + 9c_2 + \left(1 - \frac{p_2}{p_1}\right)^9 s_{2,3} + \sum_{j=1}^{9} \binom{9}{j} \left(\frac{p_2}{p_1}\right)^j \left(1 - \frac{p_2}{p_1}\right)^{9-j} (s_{2,3} + j \cdot c_3) = 39.125$. On the other hand, switching directly to V_3 incurs a total cost of $s_{1,3} + 9c_3 = 46.1$. Hence, the best strategy in this case is to try voltage V_2 first and then V_3.

Intuitively, using the intermediate voltage V_2 pays off only when there are many tasks left, in which case the extra switching overhead diminishes with respect to the potential energy gained from task execution.

5.5.3 Optimality result

In this section we prove that level algorithms are dominant when switching costs are linear. The analysis so far has assumed that the voltage switching cost follows triangle inequality. However, under the special case of linear costs, i.e., $s_{\ell,h} = s_{\ell,p} + s_{p,h}$, we are able to show that there exists an algorithm that satisfies Definition 5 and which is optimal. Furthermore, the optimal voltage sequence has a much simpler structure, which helps reduce significantly the complexity of the optimal algorithm.

We first prove a simple result: when switching costs are zero, the optimal algorithm for a single task defines a level algorithm that is optimal for an arbitrary number of tasks. Intuitively, we can transform any task-by-task algorithm into a level algorithm with the same cost, because there is no overhead to switch voltages:

Lemma 11. *On a system without voltage switching costs there exists an optimal algorithm which is a level algorithm such that after each voltage:*

- *The optimal voltage to execute the remaining tasks does not depend on the number of remaining tasks, and it is the same as the optimal voltage to execute a single task;*
- *The optimal expected energy consumption is proportional to the number of remaining tasks.*

Proof. Let us consider an optimal algorithm $\mathcal{O}$ and its execution on an instance with n tasks denoted T_1, ..., T_n. We reorder the task executions performed by $\mathcal{O}$ such as all executions of T_1 are done first, then all executions of T_2 are performed, and so on. (Note that we have taken an arbitrary ordering of the tasks.) The new execution order has exactly the same cost as the previous one because switching costs are null. Also, Assumption 3 has no impact on the solution. Therefore, the optimal sequence of voltages to try for task T_i (for any value $1 \leq i \leq n$) is the sequence of voltages defined in Section 5.4, so for a single task. Let $V_{\pi(1)}$, ..., $V_{\pi(m)}$ be this sequence. Then algorithm $\mathcal{O}$, being optimal, tried for each task this sequence of voltages, in increasing voltages, until success. We finally reorder, once again at no cost, the task executions performed by algorithm $\mathcal{O}$: first all the executions at voltage $V_{\pi(1)}$, that is, the execution of all tasks at $V_{\pi(1)}$; then all executions at voltage $V_{\pi(2)}$, that is, the execution of all remaining tasks at voltage $V_{\pi(2)}$; and so on. We have, hence, defined a level algorithm following the voltage sequence defined for a single task, and whose expected energy cost is the same as that of the optimal algorithm $\mathcal{O}$. □

Theorem 19. *With linear switching costs, level algorithms are dominant.*

Proof. Let us consider any optimal algorithm $\mathcal{O}$ and an instance with n tasks. Let V_ℓ be the lowest voltage used by algorithm $\mathcal{O}$ during the entire execution. Then, the total voltage switching cost incurred by $\mathcal{O}$ is $S^{\mathcal{O}} \geq s_{k,\ell} + s_{\ell,k}$. (Note that we use $s_{k,\ell} + s_{\ell,k}$ for clarity here, to

emphasize that we switch down to V_ℓ and back to $V_k = V_{\text{TH}}$, but remember that $s_{k,\ell} = s_{\ell,k}$ by Assumption 4.)

Let us consider any level algorithm $\mathcal{L}$ that starts by switching to voltage V_ℓ. Then, whatever the voltages it uses among the voltages V_ℓ, $V_{\ell+1}$, ..., V_k, it incurs a total switching cost exactly equal to $S = s_{k,\ell} + s_{\ell,k}$, because switching costs are linear. Therefore, we can assume without loss of generality that algorithm $\mathcal{L}$ switches to all the voltages V_ℓ, $V_{\ell+1}$, ..., V_k (maybe without executing any task at some of those levels). Then, the optimization problem to solve to determine at which next voltage should the remaining tasks be executed is exactly the same as if there were no switching costs (there is no penalty incurred when an intermediate voltage is used). Then, Lemma 11 tells us not only what is the optimal level algorithm in such a case but, also, that it is optimal among all existing algorithms. Hence, the optimality of $\mathcal{L}$ among all algorithms using voltages among V_ℓ, $V_{\ell+1}$, ..., V_k. Because $\mathcal{O}$ is one of these algorithms and is optimal, we can conclude. □

Theorem 20. *To execute a set of n independent tasks on a system with k voltages and linear switching costs, the optimal solution can be obtained with complexity $O(k^2)$.*

Proof. According to Theorem 19, we only need to focus on level algorithms to obtain the optimal solution.

Suppose that a level algorithm starts executing the tasks at voltage V_h. Then, the total switching cost paid by the algorithm during the entire execution is given by $s_{k,h} + s_{h,k}$, which is fixed and does not depend on the sequence of voltages used. It remains to find the optimal expected energy consumption to execute the tasks from V_h without considering the voltage switching costs. When there are no voltage switching costs, let $\bar{E}^*_{\text{SET}}(i, V_h, V_\ell)$ and $\bar{E}_{\text{ONE}}(V_h, V_\ell)$ denote, respectively, the optimal expected energy to execute i tasks and one task by starting from voltage V_h, given that all of them have previously failed under voltage V_ℓ. We have $\bar{E}^*_{\text{SET}}(n, V_h, V_0) = n \cdot \bar{E}^*_{\text{ONE}}(V_h, V_0)$. We can then try all possible starting voltages to get the optimal expected total energy consumption as follows:

$$E^*_{\text{SET}} = \min_{1 \le h \le k} \left\{ s_{k,h} + s_{h,k} + n \cdot \bar{E}^*_{\text{ONE}}(V_h, V_0) \right\} ,$$

and the optimal starting voltage is therefore $V_{h'}$, where $h' = \arg\min_{1 \le h \le k} \left\{ s_{k,h} + s_{h,k} + n \cdot \bar{E}^*_{\text{ONE}}(V_h, V_0) \right\}$.

Lemma 11 also shows that the optimal sequence of voltages to follow is the same as the optimal sequence to execute one task without considering voltage switching costs. According to Theorem 17, this can be computed by

$$\bar{E}^*_{\text{ONE}}(V_\ell) = \min_{\ell < h \le k} \bar{E}^*_{\text{ONE}}(V_h, V_\ell) ,$$

and $\bar{V}^*_{\text{ONE}}(V_\ell) = V_{h'}$ with $h' = \arg\min_{\ell < h \le k} \bar{E}^*_{\text{ONE}}(V_h, V_\ell)$. The complexity is determined by the computation of $\bar{E}^*_{\text{ONE}}(V_h, V_\ell)$ for all $0 \le V_\ell < V_h \le V_k$, which takes $O(k^2)$ time. □

In the general case, we will have triangle inequality but not triangle equality. We have not been able to design a counter-example to the optimality of level algorithms in the general case. We therefore conjecture the dominance of level algorithms.

5.6 Simulations

In this section, we evaluate the performance of the proposed algorithms using simulations. We instantiate the performance model with two scenarios. The first scenario is based on the available data about timing error probabilities on a specific hardware [51], and we consider a set of matrix products using ABFT as the verification mechanism. In the second scenario, we use synthetic data to assess the impact of different parameters (e.g., verification cost, error probability, switching cost) on alternative hardware and applications.

5.6.1 Comparing algorithms

We compare our dynamic programming algorithm designed for a set of independent tasks, which we denote by *DP-indep*, to the following algorithms in the evaluation.

- *Baseline* & *Threshold*: These two are static algorithms that use the *environmental margin* voltage and *nominal* voltage, respectively, to execute the tasks. The former does not make use of any voltage scaling technique, and the latter scales the voltage using near-threshold computing. Both algorithms do not incur errors and hence do not require verification and re-execution.
- *DP-single*: This algorithm uses the dynamic programming solution for a single task and applies the optimal sequence of voltages to execute all tasks in the set one after another, using the output voltage of the last task as the input voltage for the current task.

5.6.2 Matrix Multiplication on FPGA

In the first evaluation scenario, we consider a concrete application (matrix multiplication) executed on a specific platform, where error probabilities are known from real measurements.

Platform setting

We adopt the set of voltages and error probabilities measured by Ernst et al. [51] on an FPGA multiplier block. Figure 5.1 shows the error probability $p_\ell^{(1)}$ of each available voltage V_ℓ when performing a *single* operation with random inputs. We take the *zero margin* 1.54V as the nominal voltage and consider the *environmental margin* 1.69V as the base operating voltage. As in [W6], we scale the error probabilities down by a factor of 10 to account for the circuit-level error recovery technique [51]. Since the dynamic power consumption is a quadratic function of the operating voltage [22, 83], for a given voltage V_ℓ, the energy consumed to execute a task (one matrix product) is modeled as $c_\ell = V_\ell^2 w$, where w denotes the total number of operations in the task. The energy to switch the operating voltage is assumed to be linear (thus following triangle equality), and it is modeled as $s_{\ell,h} = \beta \cdot \frac{|V_\ell - V_h|}{V_k - V_1}$, where β captures the relative cost of voltage switching in comparison to computation.

Application modeling

We consider the computation of a set of matrix products of the same size, which forms a set of independent tasks of same computation cost. Each product consist of m^3 multiply-add operations, where m denotes the size of the matrices. To detect errors we employ Algorithm-Based Fault Tolerance (ABFT) [63], which uses checksums to detect, locate and even correct errors in many linear algebra kernels. Specifically, by adding one (column or row) checksum to each of the input matrices, the technique enables to detect and correct up to one error during the computation of a matrix product with an overhead of $O(m^2)$ additional operations. This is almost negligible compared to the $O(m^3)$ operations incurred by the raw computation for reasonable matrix sizes. In the simulation, we fix the matrix size to be $m = 64$, so the ABFT version has $w = m(m+1)^2 = 64 \times 65^2$ operations, incurring an overhead of about 3%. With the ability to correct one error, the probability of having an incorrect product using voltage V_ℓ is thus given by $p_\ell = 1 - \left(1 - p_\ell^{(1)}\right)^w - \binom{w}{1}\left(1 - p_\ell^{(1)}\right)^{w-1} p_\ell^{(1)}$.

Results

Figure 5.2 presents the impact of the number of tasks n on the expected energy consumptions when the voltage switching cost β is set to be equivalent to multiplying two matrices of size 64×64, which is almost the cost of one task. When the number of tasks is small (e.g., less than 7), the switching cost can not be amortized and the best choice is to stay at nominal voltage. Additionally, the cost of the verification mechanism (ABFT) when $m = 64$ reaches 3% of the total work and both *DP-single* and *DP-indep* remain worse than executing all tasks at nominal voltage and without any verification mechanism. However, when the number of tasks is large enough (e.g., more than 7), *DP-indep* quickly outperforms *DP-single*, which only focuses on one task at a time and is thus unable to lower the voltage if it is not worth it for at least one task. On the contrary, *DP-indep* is paying the switching cost only once and it is easily amortized over the execution of the entire set of tasks.

Figure 5.3 shows the impact of the switching cost β on the expected energy consumption of *DP-single* and *DP-indep* under different switching costs when the number of tasks is fixed to 32. Again, we model the switching cost β to be equivalent to multiplying two matrices of size $x \times x$. The x axis shows the corresponding matrix size. When the switching cost is small (e.g., $x = 20$), both *DP-single* and *DP-indep* yield the same expected energy consumption. In fact, they are both able to amortize the switching cost with one task and they use the same sequence of voltages. But as the switching cost increases, *DP-single* quickly shows its limits while *DP-indep* manages to better amortize the overhead and remains better than *Threshold* even when the switching cost is more than three times the cost of one task ($x > 96$). Note that when the switching cost is high enough (e.g., $x > 116$), both algorithms are unable to perform better than *Threshold*.

Overall, when the number of tasks is large enough, or if the switching cost is small, *DP-indep* is always better and it saves up to 23% of the expected energy compared to the baseline algorithm in this configuration.

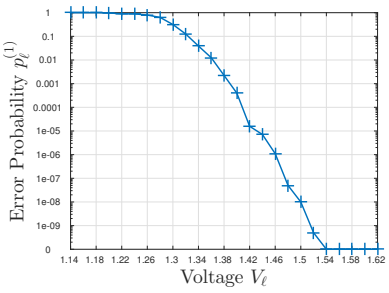


Figure 5.1: Error probabilities of available voltages measured on an FPGA multiplier block, for a single operation with random inputs [51].

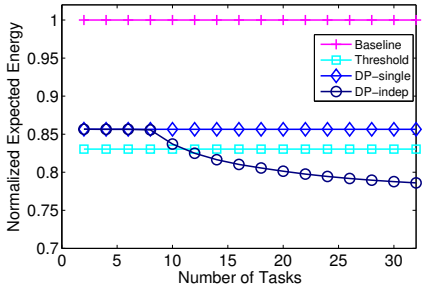


Figure 5.2: Impact of the switching cost β on the energy consumption of the algorithms.

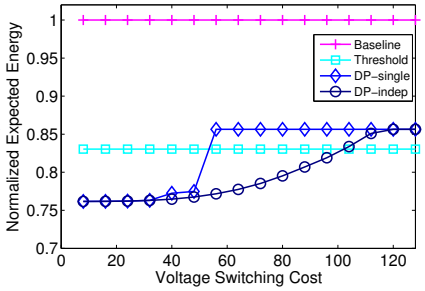


Figure 5.3: Impact of the number of tasks n on the energy consumption of the algorithms.

5.6.3 Synthetic data

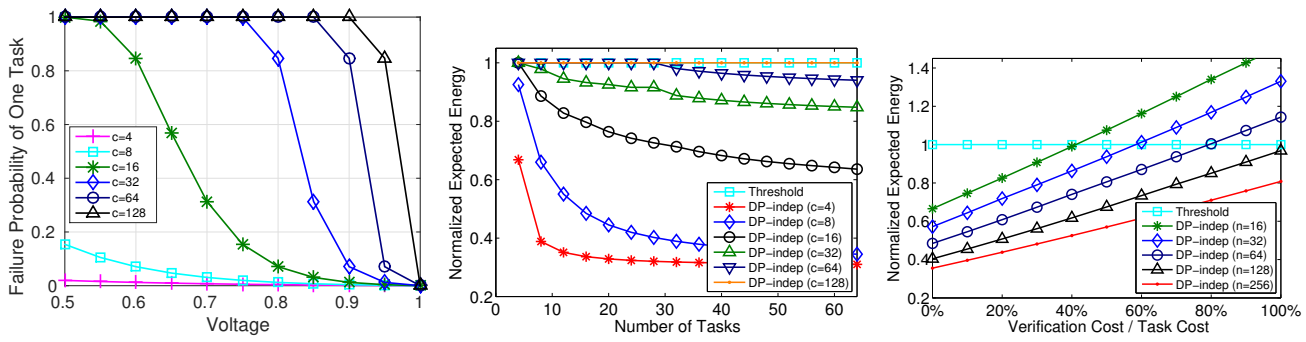
We now consider synthetic input data in order to evaluate the algorithms on different hardware and applications. We envision platforms with normalized voltages falling in [0.5, 1], where 1 represents the nominal voltage V_k. Following the characteristics of soft errors [54], we model the error probability of a computation of length w executed using voltage V_ℓ to be $p_\ell = 1 - e^{-\lambda_\ell w}$, where $\lambda_\ell = \lambda_0 e^{-c(V_\ell - V_k)} - \lambda_0$ represents the error rate and c denotes the failure rate coefficient that depends on the hardware. The model specifies the base error rate λ_0, which is usually very small. In the experiments, λ_0 is fixed at $10^{-5}s$ (less than one error per day). The energy consumption to execute tasks and voltage switching costs follow the same model as in the previous scenario.

Results

For each experiment, the total work is fixed to $W = 10000$ operations and the number of tasks is fixed at $n = 32$, so that each task has about $\frac{10000}{32} \approx 312$ operations.

Figure 5.4a shows, given a voltage, the probability of failure for one task under different failure rate factors c. This factor determines how *fast* the probability of failure increases when the voltage is decreased below threshold. A small value for c shows a very optimistic configuration where lowering the voltage below threshold is possible while keeping low probabilities of failure. Such a configuration is ideal and allows us to choose amongst a wide range of voltages, thus giving the opportunity to save energy. Higher values for c shows more realistic, and also more pessimistic configurations. When c is high (e.g., 128), as soon as we lower the voltage, the probability of failure increases dramatically and the chances of success drop close to zero. In that case, *DP-indep* has no other choice but to stay at nominal voltage.

Figure 5.4b presents the impact of the number of tasks on the normalized expected energy consumption of *DP-indep* with respect to *Threshold* when the total work W is fixed to 10000 and for different values of c. When the number of task decreases, the size of the tasks increases and the probability of success for one task drops considerably. As a result, we have to use



(a) (b) (c)

Figure 5.4: Impact of failure rate factor c ((a) and (b)), and of the cost ratio of verification (c), on the expected energy consumption.

higher voltages, which consumes more energy.

The value of c determines the range of voltages that can be used safely, i.e., with low probability of failure, as shown by Figure 5.4a and it has a huge impact on the expected energy consumption. Under a small c, the algorithm can yield important energy savings compared to *Threshold* (more than 40% for 10 tasks with $c = 4$ under this configuration), while with a large c (e.g., $c = 128$) the algorithm will not be able to do better than *Threshold*.

Finally, Figure 5.4c evaluates the impact of the verification cost on the expected energy consumption of *DP-indep*. In this experiment, we make the verification cost vary with respect to the cost of one task from 0% (no verification cost) to 100% (the verification cost is equivalent to the cost of one task). When the cost of the verification increases, so does the total overhead of the computation. Consider the case where the cost of the verification is half the cost of one task (i.e., total work including verifications is now $1.5W$). Then, executing 256 tasks is 40% better than *Threshold*, whereas under the same configuration, executing only 32 tasks for the same amount of work is only 5% better than *Threshold*. Overall, we observe that the execution of a lot of small tasks is preferred over a few big tasks.

Optimal solution with two different voltages

We now consider a set of scenarios where only two voltages are available, V_1 and $V_k = V_{\text{TH}}$. As shown previously, the only decision to make is whether to switch to the lower voltage V_1, in which case a cost of $2s_{k,1}$ is incurred due to voltage switching, or to directly use the nominal voltage V_k for executing the tasks. Figure 5.5 shows the energy savings achieved by the optimal algorithm (*DP-indep*). Naturally, more energy is saved when V_1 has a smaller execution cost or a lower error probability. For any given cost and probability, the saving also increases with the number of tasks, because the cost of voltage switching can be better amortized. For the case with 10 tasks and $s_{k,1} = 5c_1$, and when the error probability of voltage V_1 is around 0.5, *DP-indep* starts to save energy when the cost c_1 drops below $c_k/3$, and it is able to save at least 40% of the energy with a cost lower than $c_k/10$.

For the same case (10 tasks and $s_{k,1} = 5c_1$), Figure 5.6 shows the performance degradation (expected execution time overhead) due to verification and re-execution. Since the results

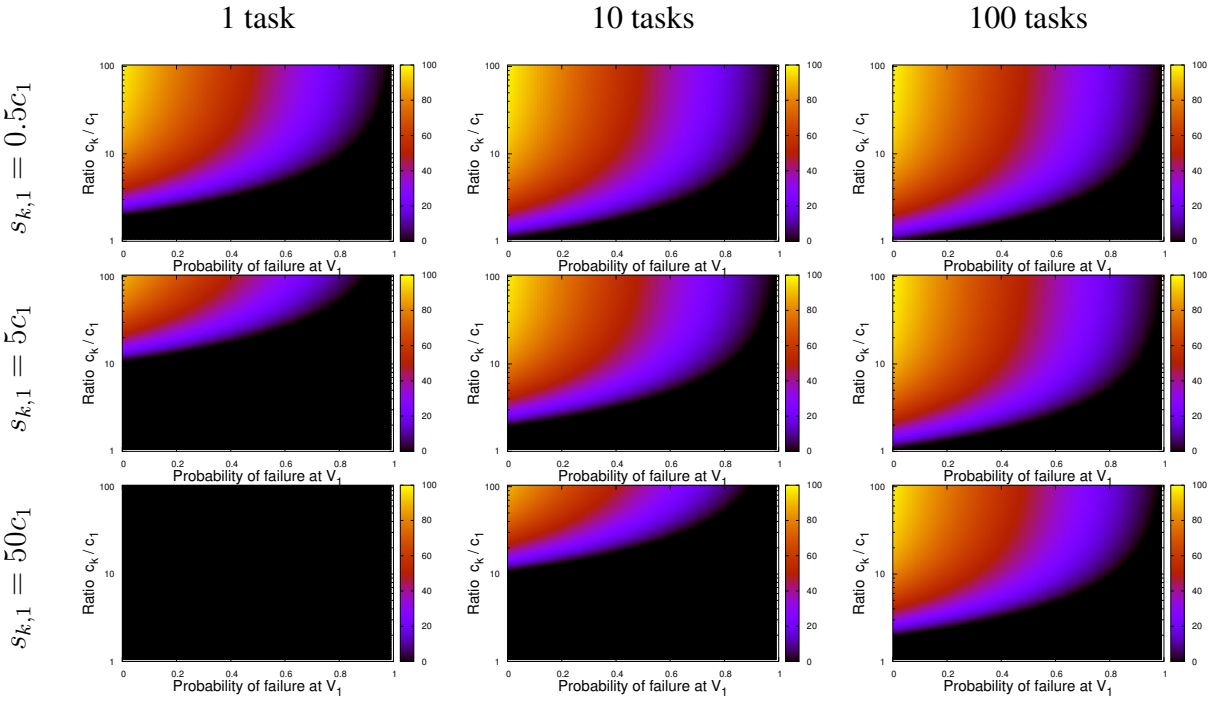


Figure 5.5: Impact of the switching cost and of the number of tasks on the percentage of energy saving when only two voltages are available. Black means no saving, and yellow means 100% of energy is saved.

are obtained for *DP-indep*, which targets energy minimization, they do not represent the best performance. Indeed, the overhead is minimum (0%) when the algorithm does not switch to V_1, which means that no energy is saved at all. Once the algorithm has decided to make the switch to save energy, the expected performance starts to degrade. For any fixed cost ratio c_k/c_1, however, a lower error probability enables the algorithm to both save energy and improve performance, because tasks enjoy a higher chance of success at V_1. Finally, as verification and/or voltage switching times increase, the performance degrades further. For instance, when $p_1 = 0.5$ and $c_k/c_1 = 10$, and when verification takes 5% of the task execution time and voltage switching takes the same time as task execution, the algorithm achieves 40% of the energy at the expense of about 70% degradation in performance.

5.7 Conclusion

In this chapter, we have used voltage overscaling to design a purely software-based approach for reducing the energy consumption of HPC applications. This approach aggressively lowers the supply voltage below the nominal voltage, introducing timing errors. Based on a formal model of timing errors, we have provided an optimal level algorithm to schedule independent tasks, and we have proven its global optimality when switching costs are linear. The evaluation results obtained both for matrix multiplication on FPGA and for synthetic data demonstrate that our approach can indeed lead to significant energy savings compared to the standard algorithm

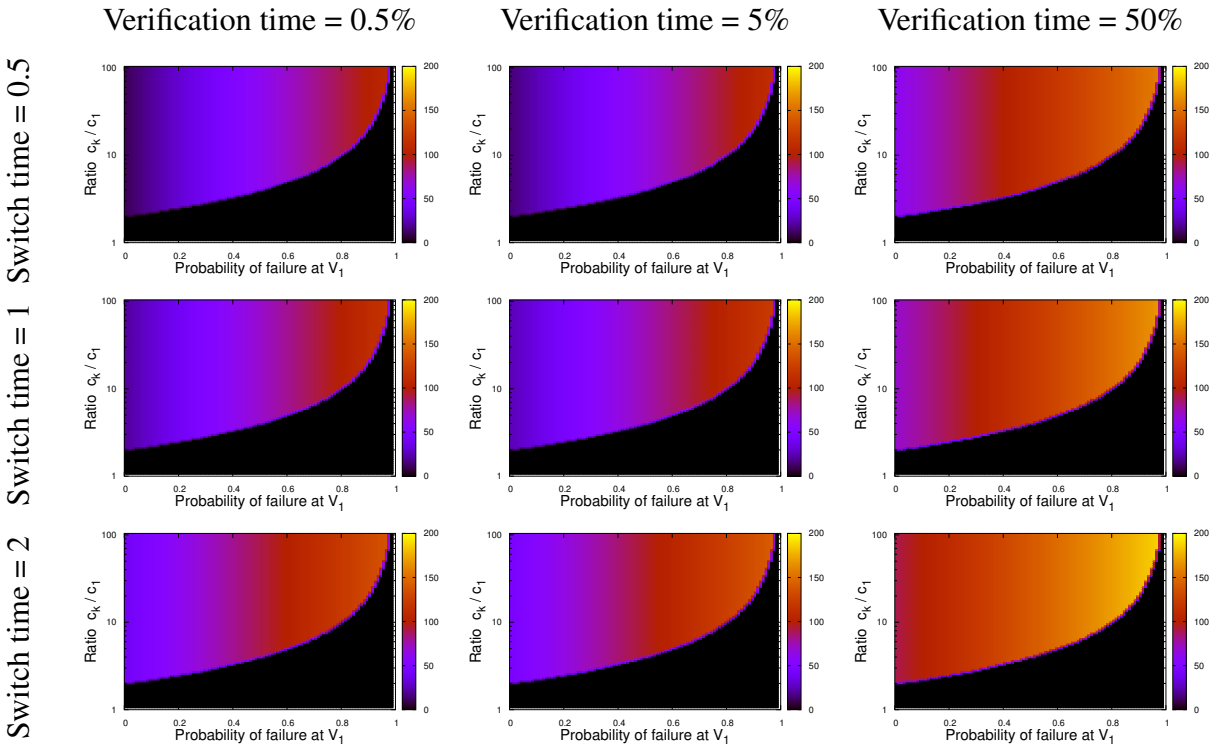


Figure 5.6: Percentage of (expected) execution time overhead as a function of the probability of failure and of the ratio of energy costs. Examples with 10 tasks and a switching energy cost of $5c_1$. The unit used to express the verification and switching times is the execution time of a task (at voltage V_k).

that always operates at (or above) the nominal voltage.

Part III

Resource Optimization

Chapter 6

When Amdahl Meets Young/Daly

This chapter investigates the optimal number of processors to execute a parallel job, whose speedup profile obeys Amdahl's law, on a large-scale platform subject to fail-stop and silent errors. Without errors, although the speedup is bounded, there is no optimal number of processors: using extra processors will always, even so slightly, benefit the performance. With errors however, adding of processors has the effect of decreasing the platform MTBF (see Equation (1)). We combine the traditional checkpointing and rollback recovery strategies with verification mechanisms to cope with both error sources. We provide an exact formula to express the execution overhead incurred by a periodic checkpointing pattern of length T and with P processors, and we give first-order approximations for the optimal values T^* and P^* as a function of the individual processor failure rate λ_{ind}. A striking result is that P^* is of the order $\lambda_{\text{ind}}^{-1/4}$ if the checkpointing cost grows linearly with the number of processors, and of the order $\lambda_{\text{ind}}^{-1/3}$ if the checkpointing cost stays bounded for any P. We conduct an extensive set of simulations to support the theoretical study. The results confirm the accuracy of first-order approximation under a wide range of parameter settings. This work has been published in the proceedings of *Cluster* [C3].

6.1 Introduction

Consider a typical HPC (High Performance Computing) application that will run for days or even weeks on a parallel platform, and whose sequential time is non-negligible. What is the optimal number of processors to execute this application so as to minimize its total execution time? Assume that the application speedup profile obeys *Amdahl's law* [1]: a fraction α of the work is sequential, while the remaining $1-\alpha$ fraction is perfectly parallel. The speedup with P processors is then

$$S(P) = \frac{1}{\alpha + \frac{1-\alpha}{P}} \, . \tag{6.1}$$

While $S(P)$ is bounded above by $1/\alpha$, it is a strictly increasing function of P, which means that one should enroll as many processors as possible to minimize execution time.

However, this reasoning only holds for error-free execution. With 100,000+ nodes in current petascale platforms, and even more computing resources when entering the exascale era,

resilience becomes a challenge [25]. Even if each node provides an individual MTBF (Mean Time Between Failures) of, say, one century, a machine with 100,000 such nodes will encounter a failure every 9 hours on average, which is smaller than the execution time of many HPC applications. Furthermore, a one-century MTBF per node is an optimistic figure, given that each node may be composed of tens or even hundreds of cores. Moreover, several types of errors need to be considered when computing at scale. In addition to the classical fail-stop errors (such as hardware failures), silent errors (or SDC, for Silent Data Corruptions) constitute another threat [74, 76, 103]. This phenomenon is not so well understood, but has been recently identified as one of the major challenges towards exascale [25].

While checkpoint/restart [28, 46, 62] is the de-facto recovery technique for dealing with fail-stop errors, there is no widely adopted general-purpose technique to cope with silent errors. In contrast to a fail-stop error whose detection is immediate, a silent error is identified only when the corrupted data leads to an unusual application behavior. Such a detection latency raises a new challenge: if the error struck before the last checkpoint, and is detected after that checkpoint, then the checkpoint is corrupted and cannot be used for rollback. In order to avoid corrupted checkpoints, an effective approach consists in employing some verification mechanism and combining it with checkpointing. Such a verification mechanism can be general-purpose (e.g., based on replication [53] or even triplication [72]) or application-specific [14, 16, 30, 85].

We address both fail-stop and silent errors by using *verified checkpoints*, which corresponds to performing a verification just before taking each checkpoint. Note that this approach is agnostic of the nature of the verification mechanism. If the verification succeeds, then one can safely store the checkpoint. Otherwise, it means that a silent error has struck since the last checkpoint, which was duly verified, and one can safely recover from that checkpoint to resume the execution of the application. Of course, if a fail-stop error strikes, we can also safely recover from the last checkpoint, just as in the classical checkpoint and rollback recovery method. We refer to this protocol as the VC protocol, and it basically amounts to replacing the cost C of a checkpoint by the cost $V + C$ of a verification followed by a checkpoint. However, because we deal with two sources of errors, one detected immediately and the other only when we reach the verification, the analysis of the optimal checkpointing strategy is more involved.

This work shows that on failure-prone platforms, it is no longer true that enrolling more processors will always decrease the (expected) execution time of a parallel application. First, more processors means more failures: if the failure rate of an individual processor is λ_{ind} (and its MTBF is $\mu_{\text{ind}} = 1/\lambda_{\text{ind}}$), then the failure rate for a platform with P processors is $P\lambda_{\text{ind}}$ (and its MTBF is μ_{ind}/P) [62, Proposition 1.2]. Second, the cost of checkpointing may well increase linearly with P [48, 101], because of the synchronization needed among the processors in order to take a coherent snapshot of the global application state. The intuition is that at some point adding more resources will be an overkill, for failures and resilience operations to handle them will become too frequent for the application to make any progress.

These considerations raise the following fundamental question: *What is the optimal number of processors to execute a parallel application on a failure-prone platform?* Surprisingly, this question has never received a quantitative answer, although some experimental study has been reported [65, 101]. The major contribution of this chapter is to answer this question by providing a detailed analysis on the performance of the VC protocol in the presence of both fail-stop

and silent errors. In particular, we consider a periodic checkpointing pattern PATTERN(T, P), which consists of a work chunk of duration T and executed with P processors, followed by a verification and then by a checkpoint (see Figure 6.1). We give first-order approximations for the optimal values T^* and P^* as a function of the individual processor failure rate λ_{ind}. A striking result is that, as long as the sequential fraction α of the application is a non-negligible constant, P^* is of the order $\lambda_{\text{ind}}^{-1/4}$ if the checkpointing cost grows linearly with the number of processors, and of the order $\lambda_{\text{ind}}^{-1/3}$ if the checkpointing cost stays bounded for any P. The corresponding values of T^* in these two cases are of the orders $\lambda_{\text{ind}}^{-1/2}$ and $\lambda_{\text{ind}}^{-1/3}$, respectively. The results nicely extend the well-known Young/Daly formula [36, 97] by characterizing the optimal number of resources to enroll. These first-order bounds are well corroborated and validated by our simulation study conducted using real platform parameters.

The main contributions of this chapter are the following:

- The derivation of an exact analytical formula for the expected execution time of a pattern in the presence of both fail-stop and silent errors, where fail-stop errors can strike at any time (while silent errors only strike during computations);
- The determination of the optimal pattern length and processor count, up to the first-order term. Given error rates and checkpoint/verification costs, we compute both the optimal pattern length and optimal number of processors to enroll. To the best of our knowledge, this is the first analytical characterization of the optimal degree of parallelism for executing a parallel application whose speedup obeys Amdahl's law;
- An extensive set of simulations with data collected from real platforms. The results confirm the accuracy of the performance model and validity of first-order approximation under a wide range of parameter settings and resilience scenarios.

The rest of the chapter is organized as follows. Section 6.2 briefly discusses the related work. Section 6.3 introduces the models and notations. Section 6.4 presents all our analytical results, followed by the presentation of the simulation results in Section 6.5. Finally, Section 6.6 provides concluding remarks and hints for future directions.

6.2 Related work

Checkpointing. The most commonly deployed strategy to cope with fail-stop errors is checkpointing, in which processes periodically save their states, so that computation can be resumed from that point when some failure disrupts the execution. Checkpointing strategies are numerous, ranging from fully coordinated checkpointing [28] to uncoordinated checkpointing and recovery with message logging [46]. Despite a very broad applicability, these fault-tolerance methods suffer from the intrinsic limitation that both protection and recovery generate an I/O workload, which grows with failure probability and becomes unsustainable at large scale [17, 52] (even with optimizations such as diskless or incremental checkpointing [78]). To reduce the checkpointing overhead, many authors have proposed multi-level checkpointing protocols, which combine global disk checkpointing with local or in-memory checkpointing [12, 37, 74, 88, 94], including the work presented in Chapter 2 and 3.

The cost of checkpointing clearly depends upon the protocol and storage type, hence we adopt a quite general formula to account for checkpoint overhead in this work. We let $C_P = a+b/P+cP$ to model the time to save a checkpoint on P processors. Here, $a+b/P$ represents the I/O overhead to write the application's memory footprint M to the storage system. For in-memory checkpointing [42, 99], $a + b/P$ is a communication time with latency a and $b/P = M/(\tau_{net}P)$, where τ_{net} is the network bandwidth (each processor stores M/P data items). For coordinated checkpointing to stable storage, there are two cases: if the storage system's bandwidth is the I/O bottleneck, $a = \beta + M/\tau_{io}$ and $b = 0$, where β is a start-up time and τ_{io} is the I/O bandwidth; otherwise, if the network is the I/O bottleneck, we retrieve the same formula as for in-memory checkpointing. Finally, cP represents the message passing overhead that grows linearly with the number of processor, in order for all processors to reach a global consistent state [48, 101].

Silent error detection. Considerable efforts have been directed at verification techniques to reveal silent errors. A perfect verification is often only achievable with expensive techniques, such as process replication [53, 75] or redundancy [45, 72]. Application-specific information can be very useful in decreasing the verification cost. Algorithm-based fault tolerance (ABFT) [16, 63, 87] is a well-known technique to detect errors in linear algebra kernels using checksums. Various techniques have been proposed in other application domains. Benson et al. [14] compared a higher-order scheme with a lower-order one to detect errors in the numerical analysis of ODEs. Chen [30] uses orthogonality checks for Krylov-based sparse solvers. Sao and Vuduc [85] investigate self-stabilizing corrections after error detection in the conjugate gradient method. Heroux and Hoemmen [19] design a fault-tolerant GMRES capable of converging despite silent errors. Bronevetsky and de Supinski [21] provide a comparative study of detection costs for iterative methods. Recently, detectors based on data analytics, using use interpolation techniques, such as time series prediction and spatial multivariate interpolation, have also been proposed as verification mechanisms [8, 10, 15]. Altogether, there is a wide range of available detectors, and our approach is agnostic of the nature of verification mechanism used for silent errors.

Resilience and speedup. Several authors have investigated the optimal number of processors to enroll when running a parallel application on a failure-prone platform. Zheng et al. [101] address this problem for fail-stop errors and provide a formula to compute the speedup of an application obeying Amdhal's law and running with P processors. Also for fail-stop errors, Jin et al. [65] use an iterative relaxation procedure to compute the optimal number of resources for a perfectly parallel job. These two important works are the most closely related to ours. In comparison, the main differences with our work are: (i) we account for both fail-stop and silent errors (instead of only fail-stop errors); (ii) we consider several relevant scenarios for checkpointing costs (instead of only linearly growing costs); (iii) we analytically characterize both the optimal number of processors and optimal checkpointing period as a function of the individual processor failure rate, the speedup profile and the checkpoint/verification cost (instead of using numerical procedures); and (iv) our formulas are exact up to first-order term and account for errors in checkpointing. Our first-order approximation formulas (see Theorems 22 and 23 below) are the first quantitative assessments of the best degree of parallelism that should be deployed.

6.3 Models and notations

This section presents the analytical models for evaluating the performance of resilience algorithms. Table I summarizes the list of main notations used in the chapter.

Table I
LIST OF NOTATIONS.

Application parameters	
PATTERN(T, P)	Periodic checkpointing pattern
T	Length (or period) of pattern
P	Number of allocated processors
$S(P)$	Speedup function w/o failure
$H(P) = 1/S(P)$	Execution overhead w/o failure
$\mathbb{E}$(PATTERN) or $\mathbb{E}(T, P)$	Expected exec. time of a pattern
$\mathbb{S}$(PATTERN) or $\mathbb{S}(T, P)$	Expected speedup of a pattern
$\mathbb{H}$(PATTERN) or $\mathbb{H}(T, P)$	Expected exec. overhead of a pattern
Resilience parameters	
$\lambda_{\text{ind}} = 1/\mu_{\text{ind}}$	Error rate of an individual processor
$\lambda_P^f = f\lambda_{\text{ind}}P$	Fail-stop error rate on P processors
$\lambda_P^s = s\lambda_{\text{ind}}P$	Silent error rate on P processors
$C_P = a + b/P + cP$	Checkpointing cost on P processors
$R_P = a + b/P + cP$	Recovery cost on P processors
$V_P = v + u/P$	Verification cost on P processors
D	Downtime after a fail-stop error

Failure model. We incorporate both hardware faults and silent data corruptions, which are also known as *fail-stop errors* and *silent errors* in the literature. Since the two types of errors are caused by different sources on realistic systems, we assume that they are independent and that both arrivals follow *exponential* distributions. Let $\lambda_{\text{ind}} = 1/\mu_{\text{ind}}$ denote the reciprocal of the MTBF μ_{ind} of each individual processor by accounting for both types of errors, and suppose f fraction of the total number of errors are fail-stop and the remaining $s = 1 - f$ fraction are silent. Then, the arrival rates of fail-stop and silent errors when using P processors are given by $\lambda_P^f = f\lambda_{\text{ind}}P$ and $\lambda_P^s = s\lambda_{\text{ind}}P$, respectively [62]. Thus, the probability of encountering at least one fail-stop error during a computation of time T is $q_P^f(T) = 1 - e^{-\lambda_P^f T}$ and that of encountering at least one silent error during the same computation is $q_P^s(T) = 1 - e^{-\lambda_P^s T}$.

Application model. We consider HPC applications that are long-lasting even when executed on a large number of processors. Suppose an application has a total amount of computation (or work) W_{total} and a speedup function $S(P)$ when executed on P processors without considering failures. In this chapter, we consider the speedup function obeying Amdahl's law as given in Equation (6.1). For convenience, we define $H(P) = \frac{1}{S(P)} = \alpha + \frac{1-\alpha}{P}$ to be the execution overhead of the application, where α denotes the fraction of the application that is inherently sequential and cannot be parallelized. The makespan (total execution time) W_{final} of the application in an error-free execution is therefore given by $W_{\text{final}} = \frac{W_{\text{total}}}{S(P)} = H(P)W_{\text{total}}$.

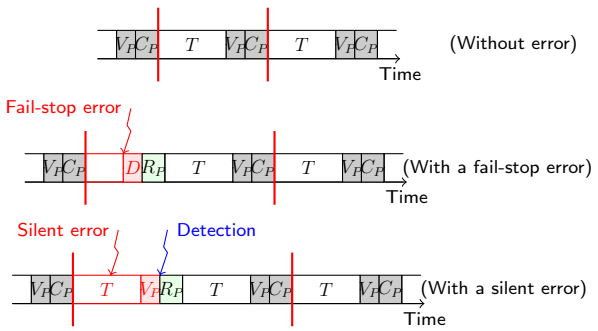


Figure 6.1: Illustration of a resilience protocol using periodic checkpointing patterns (highlighted in red). The first figure shows the execution of a pattern without any error. The second figure shows that the execution is stopped immediately when a fail-stop error strikes, in which case the pattern is re-executed after a downtime and a recovery. The third figure shows that the execution continues when a silent error strikes, till the error is detected by the verification at the end. The pattern is then re-executed after a recovery.

Resilience model. To enforce resilience, a standard protocol is by checkpointing the status of the application periodically, thus creating periodic checkpointing *patterns* as illustrated in Figure 6.1. Following the approach of Chapter 2, an additional error detection (or verification) mechanism is performed just before taking each checkpoint. If a fail-stop error strikes inside a pattern, the computation is interrupted immediately, while a silent error, if strikes, is only detected at the end of the pattern by the verification. In both cases, we roll back to the beginning of the pattern and recover from the last checkpoint, thus avoiding restarting the application from scratch. Note that a fail-stop error could strike after a silent error within the same pattern but before the verification is reached. In this case, the silent error is masked by the fail-stop error and need not be detected, since a recovery is nevertheless required.

Formally, we characterize a periodic checkpointing pattern, denoted as $\textsc{Pattern}(T, P)$, by the following two parameters.

- T: length (or period) of the pattern, i.e., amount of time to do useful computation before taking each checkpoint;
- P: number of processors allocated to the application.

As discussed in Section 6.2, we let $C_P = a + b/P + cP$ denote the time to save a checkpoint on P processors. The recovery cost is assumed to be the same as the checkpointing cost, i.e., $R_P = C_P$, because it involves the same I/O operations. To perform a verification, we assume the use of application-specific error detection techniques (as detailed in Section 6.2). Since a verification is only done in memory, its cost can be modeled as $V_P = v + u/P$. Here, v is a start-up overhead, and u/P is the time needed to verify the application data distributed across P processors. Finally, a constant downtime D is required after each fail-stop error in order to replace or repair a failed processor.

In our analysis, fail-stop errors can strike at any time during the execution of an application, including verifications, checkpointing and recoveries. However, silent errors can only strike

the computations, since otherwise they cannot be detected. Hence, we assume that I/O operations and verifications are protected from silent errors (e.g., by using expensive redundancy or replication techniques). Finally, no error of any kind can strike during downtime.

Optimization objective. The objective is to minimize the expected total execution time (or makespan) of an application. Since the application is divided into periodic checkpointing patterns defined by $\text{PATTERN}(T, P)$, the amount of work done in a pattern is $W_{\text{pattern}} = T \cdot S(P)$. For long-lasting applications, the total number of patterns in the application can be approximated as $\frac{W_{\text{total}}}{W_{\text{pattern}}} = \frac{W_{\text{total}}}{T \cdot S(P)}$. Let $\mathbb{E}(\text{PATTERN})$ denote the expected execution time of the pattern. The expected makespan $\mathbb{E}(W_{\text{final}})$ of the application is then given by $\mathbb{E}(W_{\text{final}}) \approx \mathbb{E}(\text{PATTERN}) \frac{W_{\text{total}}}{T \cdot S(P)}$. Now, define $\mathbb{S}(\text{PATTERN}) = \frac{T \cdot S(P)}{\mathbb{E}(\text{PATTERN})}$ to be the expected speedup of the pattern, and define $\mathbb{H}(\text{PATTERN}) = \frac{1}{\mathbb{S}(\text{PATTERN})} = \frac{\mathbb{E}(\text{PATTERN})}{T} H(P)$ to be the expected execution overhead of the pattern. The expected makespan of the application can therefore be written as $\mathbb{E}(W_{\text{final}}) \approx \frac{W_{\text{total}}}{\mathbb{S}(\text{PATTERN})} = \mathbb{H}(\text{PATTERN}) W_{\text{total}}$. We observe that the optimal expected makespan is obtained by maximizing the expected speedup or minimizing the expected execution overhead of a periodic checkpointing pattern. In the next section, we will focus on such a pattern $\text{PATTERN}(T, P)$, and find its optimal length T and processor count P.

6.4 Optimal periodic checkpointing pattern

In this section, we analytically determine the optimal periodic checkpointing pattern using first-order approximation, and derive explicit formulas for the checkpointing period T and processor allocation P. We validate the first-order solution in Section 6.5 by showing its close proximity to the optimal numerical solution.

6.4.1 Expected execution time of a pattern

We start by computing the expected execution time of a pattern when the parameters T and P are given.

Proposition 10. *The expected execution time of a given pattern* $\text{PATTERN}(T, P)$ *is*

$$\mathbb{E}(\text{PATTERN}) = \left(\frac{1}{\lambda_P^f} + D \right) \left(e^{\lambda_P^f C_P} \left(1 - e^{\lambda_P^s T} \right) + e^{\lambda_P^f R_P} \left(e^{\lambda_P^f (C_P + T + V_P) + \lambda_P^s T} - 1 \right) \right). \tag{6.2}$$

Proof. To successfully execute a pattern $\text{PATTERN}(T, P)$, we need to complete the pattern length T, the verification V_P and the checkpoint C_P. Hence, according to the linearity of expectation, we have

$$\begin{aligned} \mathbb{E}(\text{PATTERN}) &= \mathbb{E}(T + V_P + C_P) \\ &= \mathbb{E}(T + V_P) + \mathbb{E}(C_P) \,, \end{aligned} \tag{6.3}$$

We first compute $\mathbb{E}(C_P)$, the expected time to successfully store a checkpoint subject to fail-stop errors. During checkpointing, there is a probability $q_P^f(C_P)$ that a fail-stop error strikes. If that happens, we need to perform a recovery from the last checkpoint after a downtime, and then re-execute both T and V_P before re-executing C_P again. If there is no error, we just need to pay the checkpointing cost C_P. Therefore, we can express $\mathbb{E}(C_P)$ as

$$\begin{aligned}\mathbb{E}(C_P) = q_P^f(C_P)\left(\mathbb{E}^{\text{lost}}(C_P) + D + \mathbb{E}(R_P) + \mathbb{E}(T+V_P) + \mathbb{E}(C_P)\right) \\ + \left(1 - q_P^f(C_P)\right)C_P \,, \end{aligned} \tag{6.4}$$

where $\mathbb{E}(R_P)$ denotes the expected time to perform a recovery, and $\mathbb{E}^{\text{lost}}(C_P)$ denotes the expected time lost executing C_P if a fail-stop error strikes. More generally, we can define $\mathbb{E}^{\text{lost}}(W)$ to be the expected time lost for any execution of length W, and it can be computed as follows:

$$\mathbb{E}^{\text{lost}}(W) = \int_0^\infty t\mathbb{P}(X = t | X < W)dt = \frac{\int_0^W t\lambda_P^f e^{-\lambda_P^f t}dt}{\mathbb{P}(X < W)} \,,$$

where $\mathbb{P}(X = t)$ denotes the probability that a fail-stop error strikes exactly at time t. By definition, we have $\mathbb{P}(X < W) = q_P^f(W) = 1 - e^{-\lambda_P^f W}$. Integrating by parts, we get

$$\mathbb{E}^{\text{lost}}(W) = \frac{1}{\lambda_P^f} - \frac{W}{e^{\lambda_P^f W} - 1} \,. \tag{6.5}$$

Now, substituting $q_P^f(C_P)$ and $\mathbb{E}^{\text{lost}}(C_P)$ into Equation (6.4), we can get

$$\mathbb{E}(C_P) = \left(e^{\lambda_P^f C_P} - 1\right)\left(\frac{1}{\lambda_P^f} + D + \mathbb{E}(R_P) + \mathbb{E}(T+V_P)\right) \,.$$

Now, we compute $\mathbb{E}(R_P)$, the expected time to successfully perform a recovery subject to fail-stop errors. Unlike checkpointing, a recovery is always done at the beginning of a pattern. With probability $q_P^f(R_P)$, it fails due to a fail-stop error and we have to try again after a downtime. Otherwise, we just need to pay the recovery cost R_P. Therefore, we have

$$\begin{aligned}\mathbb{E}(R_P) = q_P^f(R_P)\left(\mathbb{E}^{\text{lost}}\left(R_P\right) + D + \mathbb{E}\left(R_P\right)\right) \\ + \left(1 - q_P^f(R_P)\right)R_P \,, \end{aligned}$$

which leads to

$$\mathbb{E}(R_P) = \left(\frac{1}{\lambda_P^f} + D\right)\left(e^{\lambda_P^f R_P} - 1\right) \,.$$

In order to compute $\mathbb{E}(\text{PATTERN})$, and according to Equation (6.3), we need to compute $\mathbb{E}(T + V_P)$. Recall that fail-stop errors can strike at any time during the execution, while silent errors only strike during computations. When a fail-stop error strikes, which happens with probability $q_P^f(T + V_P)$, we do not need to account for silent errors, since the application is stopped immediately and we need to re-execute $T + V_P$ anyway, following a downtime and a

recovery. Otherwise, with probability $1-q_P^f(T+V_P)$, there is no fail-stop error, and only in this case, we check for silent errors. With probability $q_P^s(T)$, a silent error strikes (and is detected by the verification), and we need to perform a recovery and re-execute $T + V_P$. Otherwise, the execution is complete. Overall, we have

$$\begin{aligned}\mathbb{E}(T+V_P) = q_P^f(T+V_P)\left(\mathbb{E}^{\text{lost}}(T+V_P) + D + \mathbb{E}(R_P) + \mathbb{E}(T+V_P)\right) \\ + \Big(1-q_P^f(T+V_P)\Big)\Big(T+V_P+q_P^s(T)\big(\mathbb{E}(R_P)+\mathbb{E}(T+V_P)\big)\Big).\end{aligned}$$

Plugging $q_P^f(T+V_P)$, $q_P^s(T)$ and $\mathbb{E}^{\text{lost}}(T+V_P)$ into the above equation, and solving for $\mathbb{E}(T+V_P)$, we can get

$$\begin{aligned}\mathbb{E}(T+V_P) = e^{\lambda_P^s T}\big(e^{\lambda_P^f(T+V_P)}-1\big)\left(\frac{1}{\lambda_P^f}+D\right) \\ + e^{\lambda_P^s(T+V_P)}\big(T+V_P\big) \\ + \big(e^{\lambda_P^f(T+V_P)+\lambda_P^s(T)}-1\big)\mathbb{E}(R_P)\,.\end{aligned}$$

Finally, plugging $\mathbb{E}(T+V_P)$, $\mathbb{E}(C_P)$ and $\mathbb{E}(R_P)$ back into Equation (6.3), we find that

$$\begin{aligned}\mathbb{E}(\textsc{Pattern}) = \left(\frac{1}{\lambda_P^f}+D\right)\Bigg(e^{\lambda_P^f C_P+\lambda_P^s T}\big(e^{\lambda_P^f(T+V_P)}-1\big) \\ + e^{\lambda_P^f C_P}-1+\big(e^{\lambda_P^f R_P}-1\big)\left(e^{\lambda_P^f(T+V_P+C_P)+\lambda_P^s(T)}-1\right)\Bigg),\end{aligned}$$

which simplifies to the expression shown in Equation (6.2). □

To find the optimal pattern, one needs to search for values of T and P that minimize the expected execution overhead $\mathbb{H}(\textsc{Pattern})$ of a pattern based on the expected execution time $\mathbb{E}(\textsc{Pattern})$ computed above. However, due to the complex expression given by Equation (6.2), an analytical solution is difficult to find, and one has to rely on numerical methods to approximate the optimal solution. In the following, we will use first-order approximation to derive explicit formulas for the optimal checkpointing period and processor allocation. The simulation results in Section 6.5 show that first-order approximation offers very close estimates to the optimal solution.

6.4.2 Limitation of first-order approximation

Before deriving the optimal pattern parameters, we first investigate the limitation of first-order approximation by bounding the maximum orders of T and P that can be approximated by the approach. Suppose P and T satisfy

$$P = \Theta(\lambda_{\text{ind}}^{-x}) \text{ and } T = \Theta(\lambda_{\text{ind}}^{-y})\,,$$

where $x, y > 0$. Since $\lambda_P C_P = \lambda_{\text{ind}} P(a + b/P + cP)$ and $\lambda_P V_P = \lambda_{\text{ind}} P(v + u/P)$, let $\lambda_P(C_P + V_P) = \lambda_{\text{ind}} P(d + h/P + cP)$ with $d = a + v$ and $h = b + u$. Hence, we have $\lambda_P(C_P + V_P) = \Theta\left(\lambda_{\text{ind}}^{\epsilon}\right)$, where

$$\epsilon = \begin{cases} 1 - 2x & \text{if } c \neq 0 \\ 1 - x & \text{if } c = 0 \text{ and } d \neq 0 \\ 1 & \text{if } c = 0 \text{ and } d = 0 \end{cases},$$

and $\lambda_P T = \Theta(\lambda_{\text{ind}}^{1-x-y})$. Therefore, in order to accurately estimate $e^{\lambda_P C_P}$, $e^{\lambda_P V_P}$ and $e^{\lambda_P T}$ using Taylor series expansion, we need $\epsilon > 0$ and $1 - x - y > 0$, which translates to

$$x < \delta, \text{ where } \delta = \begin{cases} 1/2 & \text{if } c \neq 0 \\ 1 & \text{if } c = 0 \end{cases}, \tag{6.6}$$

$$y < 1 - x\,. \tag{6.7}$$

Inequalities (6.6) and (6.7) specify, respectively, the maximum order on the number of processors and, for a fixed processor count, the maximum order on the checkpointing period. The first-order results obtained within these bounds offer valid approximation to the optimal solution as long as the MTBF $\mu_{\text{ind}} = 1/\lambda_{\text{ind}}$ of an individual processor is sufficiently large (e.g., in the order of years, which is true for modern processors). Beyond these bounds, unfortunately, the first-order analysis will no longer be applicable.

6.4.3 Optimal checkpointing period for fixed processor count

In this section, we derive the optimal checkpointing period when the application is run with a fixed number of processors. The result extends the classical formula given by Young [97] and Daly [36] for fail-stop errors only.

Theorem 21. *Given a processor allocation $P = \Theta(\lambda_{\text{ind}}^{-x})$ with $x < \delta$ as shown in Inequality (6.6), the optimal checkpointing period of a pattern is*

$$T_P^* = \sqrt{\frac{V_P + C_P}{\frac{\lambda_P^f}{2} + \lambda_P^s}}\,. \tag{6.8}$$

The expected execution overhead (ignoring lower-order terms) in this case is given by

$$\mathbb{H}\left(T_P^*, P\right) = H(P)\left(1 + 2\sqrt{\left(\frac{\lambda_P^f}{2} + \lambda_P^s\right)(V_P + C_P)}\right). \tag{6.9}$$

Proof. For a fixed $P = \Theta(\lambda_{\text{ind}}^{-x})$ with $x < \delta$, we can consider C_P, R_P, V_P as constants, which are smaller in magnitude compared to the platform MTBFs $1/\lambda_P^f$ and $1/\lambda_P^s$. Applying Taylor

series to expand $e^z = 1+z+\frac{z^2}{2}$ up to the second-order term, we rewrite the expected execution time $\mathbb{E}(\text{PATTERN})$ of a pattern (Equation (6.2)) as follows (ignoring lower-order terms):

$$\begin{aligned}\mathbb{E}(\text{PATTERN}) = T + V_P + C_P + \left(\frac{\lambda_P^f}{2} + \lambda_P^s\right) T^2 \\ + \lambda_P^f T\left(V_P + C_P + R_P + D\right) + \lambda_P^s T\left(V_P + R_P\right) \\ + \lambda_P^f C_P \left(\frac{C_P}{2} + R_P + V_P + D\right) \\ + \lambda_P^f V_P \left(V_P + R_P + D\right) \ .\end{aligned}$$

The expected execution overhead of the pattern can then be computed as

$$\mathbb{H}(T,P) = H(P)\left(\frac{\left(V_P + C_P\right)\left(1 + O\left(\lambda_{\text{ind}}^{\epsilon'}\right)\right)}{T} + \left(\frac{\lambda_P^f}{2} + \lambda_P^s\right) T + 1 + O\left(\lambda_{\text{ind}}^{\epsilon'}\right)\right) ,$$

where $\epsilon' = 1 - 2x$ if $c \neq 0$ and $\epsilon' = 1 - x$ otherwise. Since $P = \Theta(\lambda_{\text{ind}}^{-x})$ is fixed and $x < \delta$, we have $\epsilon' > 0$ and hence the term $O\left(\lambda_{\text{ind}}^{\epsilon'}\right)$ becomes negligible (in front of 1) when λ_{ind} is sufficiently small (e.g., tends to 0). Given a processor allocation P, the optimal expected overhead is achieved by setting

$$\frac{\partial \mathbb{H}(T,P)}{\partial T} = H(P)\left(-\frac{V_P + C_P}{T^2} + \frac{\lambda_P^f}{2} + \lambda_P^s\right) = 0 ,$$

which gives rise to the optimal checkpointing period T_P^* as shown in Equation (6.8). Now, substituting T_P^* back into $\mathbb{H}(T,P)$, we obtain the expected execution overhead as shown in Equation (6.9). □

Theorem 21 shows that, for a given processor count $P = \Theta(\lambda_{\text{ind}}^{-x})$ with $x < \delta$ as specified by Inequality (6.6), the optimal checkpointing period satisfies $T_P^* = O(\lambda_{\text{ind}}^{-y})$, where

$$y = \begin{cases} 1/2 & \text{if } c \neq 0 \\ (1-x)/2 & \text{if } c = 0 \text{ and } d \neq 0 \\ 1/2 - x & \text{if } c = 0 \text{ and } d = 0 \end{cases} .$$

In all cases, we get $y < 1 - x$ as specified by Inequality (6.7), thus validating the accuracy of the first-order approximation.

Note that, in the case of $c = 0$ and $d = 0$, we also need $x < 1/2$ in order to have $y > 0$. This additional constraint on the order of P is required to derive the first-order approximation for the optimal checkpointing period as given in Equation (6.8).

6.4.4 Optimal processor allocation and pattern parameters

We now optimize the number of allocated processors to an application. We discuss different cases based on the characteristic of the error-free overhead $H(P)$, as well as on the scalability of checkpointing and verification costs, which have the general form $C_P = a + \frac{b}{P} + cP$ and $V_P = v + \frac{u}{P}$. In the following analysis, we assume that all the parameters a, b, c, v, u and the sequential fraction α are constants and independent of the error rate λ_{ind}.

$H(P) = \alpha + \frac{1-\alpha}{P}$ **and** $C_P = cP + o(P)$**,** $\alpha, c \neq 0$

This case corresponds to the application having a constant sequential fraction and a checkpointing cost that grows linearly with the number of processors (the verification cost has no impact in this scenario).

Theorem 22. *Suppose the application has a constant sequential fraction $\alpha > 0$, and a checkpointing cost $C_P = cP + o(P)$. The optimal number of processors and the corresponding optimal checkpointing period of a pattern are*

$$P^* = \left(\frac{1}{c\left(\frac{f}{2}+s\right)\lambda_{\text{ind}}}\right)^{1/4}\left(\frac{1-\alpha}{2\alpha}\right)^{1/2}, \tag{6.10}$$

$$T^* = \left(\frac{c}{\left(\frac{f}{2}+s\right)\lambda_{\text{ind}}}\right)^{1/2}. \tag{6.11}$$

The expected execution overhead (ignoring lower-order terms) in this case is

$$\mathbb{H}(T^*, P^*) = \alpha + 2\left(4\alpha^2(1-\alpha)^2 c\left(\frac{f}{2}+s\right)\lambda_{\text{ind}}\right)^{1/4}. \tag{6.12}$$

Proof. Substituting $H(P) = \alpha + \frac{1-\alpha}{P}$ into Equation (6.9) and applying $C_P + V_P = cP + o(P)$, we can get the expected execution overhead as follows:

$$\mathbb{H}\left(T_P^*, P\right) = \alpha + 2\alpha P\sqrt{c\left(\frac{f}{2}+s\right)\lambda_{\text{ind}}} + \frac{1-\alpha}{P} + o\left(\lambda_{\text{ind}}^{1/2}P\right).$$

The above overhead is minimized when setting

$$\frac{\partial \mathbb{H}\left(T_P^*, P\right)}{\partial P} = 2\alpha\sqrt{c\left(\frac{f}{2}+s\right)\lambda_{\text{ind}}} - \frac{1-\alpha}{P^2} + o\left(\lambda_{\text{ind}}^{1/2}\right) = 0.$$

Keeping only the dominating term, the equation above leads to the optimal processor allocation P^* as shown in Equation (6.10). Now, substituting P^* back into T_P^* and $\mathbb{H}\left(T_P^*, P\right)$ and simplifying, we obtain the optimal checkpointing period T^* and optimal expected execution overhead $\mathbb{H}(T^*, P^*)$ as shown in Equations (6.11) and (6.12), respectively. □

$H(P) = \alpha + \frac{1-\alpha}{P}$ **and** $C_P + V_P = d + o(1)$**,** $\alpha, d \neq 0$

This case corresponds to the application having a constant sequential fraction, and a constant checkpointing (and verification) cost.

Theorem 23. *Suppose the application has a constant sequential fraction $\alpha > 0$, and a checkpointing and verification cost $C_P + V_P = d + o(1)$. The optimal number of processors and the*

corresponding optimal checkpointing period of a pattern are

$$P^* = \left(\frac{1}{d\left(\frac{f}{2}+s\right)\lambda_{\text{ind}}}\right)^{1/3}\left(\frac{1-\alpha}{\alpha}\right)^{2/3}, \tag{6.13}$$

$$T^* = \left(\frac{d^2}{\left(\frac{f}{2}+s\right)\lambda_{\text{ind}}}\right)^{1/3}\left(\frac{\alpha}{1-\alpha}\right)^{1/3}. \tag{6.14}$$

The expected execution overhead (ignoring lower-order terms) in this case is

$$\mathbb{H}(T^*, P^*) = \alpha + 3\left(\alpha^2(1-\alpha)d\left(\frac{f}{2}+s\right)\lambda_{\text{ind}}\right)^{1/3}. \tag{6.15}$$

Proof. When $H(P) = \alpha + \frac{1-\alpha}{P}$ and $C_P + V_P = d + o(1)$, we can get from Equation (6.9) the expected execution overhead as follows:

$$\mathbb{H}\left(T_P^*, P\right) = \alpha + 2\alpha\sqrt{d\left(\frac{f}{2}+s\right)\lambda_{\text{ind}}P} + \frac{1-\alpha}{P} + o\left(\lambda_{\text{ind}}^{1/2}P^{1/2}\right).$$

Again, the overhead is minimized by setting $\frac{\partial \mathbb{H}(T_P^*,P)}{\partial P} = 0$, which gives

$$\alpha\sqrt{d\left(\frac{f}{2}+s\right)\frac{\lambda_{\text{ind}}}{P}} - \frac{1-\alpha}{P^2} + o\left(\lambda_{\text{ind}}^{1/2}P^{-1/2}\right) = 0.$$

Solving the equation above while focusing on the dominating term gives the optimal processor allocation P^* as shown in Equation (6.13). Substituting P^* back into T_P^* and $\mathbb{H}\left(T_P^*, P\right)$, we get the optimal T^* and $\mathbb{H}(T^*, P^*)$ as shown in Equations (6.14) and (6.15). □

$H(P) = \alpha + \frac{1-\alpha}{P}$ and $C_P + V_P = \frac{h}{P}$, $\alpha, h \neq 0$

This case corresponds to the application having a constant sequential fraction, and a checkpointing (and verification) cost that decreases linearly with the number of processors.

Recall in this case that the number of processors satisfies $P = \Theta(\lambda_{\text{ind}}^{-x})$ with $x < 1/2$ for the first-order approximation to be valid. Subject to this bound, the expected execution overhead as shown in Equation (6.9) becomes

$$\mathbb{H}(T_P^*, P) = \left(\alpha + \frac{1-\alpha}{P}\right)\left(1 + 2\sqrt{h\left(\frac{f}{2}+s\right)\lambda_{\text{ind}}}\right),$$

which decreases monotonically as the number of allocated processors P increases up to the order of $\lambda_{\text{ind}}^{-1/2}$. Asymptotically, the overhead satisfies $\mathbb{H}(T_P^*, P) = \alpha + \Theta\left(\lambda_{\text{ind}}^x\right)$ for $x < 1/2$.

Hence, it is better to enroll as many processors as possible in this case, as long as P is within the approximation bound of $O(\lambda_{\text{ind}}^{-1/2})$. Intuitively, the costs of both checkpointing and verification reduce with the processor count, which enables to place both resilience operators more frequently with smaller checkpointing period to compensate for the increased error rate. Numerical simulations conducted in Section 6.5 show that the optimal number of processors P^* is nevertheless bounded in this case with a value beyond $O(\lambda_{\text{ind}}^{-1/2})$.

$H(P) = \frac{1}{P}$

In this case, the application has a perfectly linear speedup function. Again, the expected execution overhead decreases monotonically with the number of allocated processors, and the following gives the expression for $\mathbb{H}(T_P^*, P)$ under different cases (with lower-order terms ignored):

$$\mathbb{H}(T_P^*, P) = \begin{cases} \frac{1}{P} + 2\sqrt{c\left(\frac{f}{2} + s\right)\lambda_{\text{ind}}} & \text{if } c \neq 0 \\ \frac{1}{P} + 2\sqrt{d\left(\frac{f}{2} + s\right)\frac{\lambda_{\text{ind}}}{P}} & \text{if } c = 0, d \neq 0 \\ \frac{1}{P}\left(1 + 2\sqrt{h\left(\frac{f}{2} + s\right)\lambda_{\text{ind}}}\right) & \text{if } c = 0, d = 0 \end{cases}.$$

In all the cases above, the overhead is asymptotically bounded by $\Theta\left(\lambda_{\text{ind}}^x\right)$ for $x < 1/2$, except in the second case (i.e., $c = 0$, $d \neq 0$) where $x < 1$. Numerical simulations conducted in Section 6.5 show that the optimal processor count P^* happens around $x = 1/2$ and $x = 1$ for case 1 and case 2, respectively, whereas it is unbounded for the last case, due to the combination of diminishing resilience cost and perfect application speedup.

6.4.5 Discussions

Consider a (standard) application that is not perfectly parallel (i.e., $\alpha \neq 0$). Theorems 22 and 23 show the impact of the checkpointing cost on the optimal degree of parallelism. When this cost grows linearly with P (e.g., with coordinated checkpointing on stable storage [28]), Theorem 22 states that the optimal number of processors is $P^* = \Theta(\lambda_{\text{ind}}^{-1/4})$. In that case, the optimal period has length $T^* = \Theta(\lambda_{\text{ind}}^{-1/2})$. But when this cost remains bounded (e.g., with in-memory checkpointing [99]), then Theorem 23 shows that the optimal solution has both increased parallelism $P^* = \Theta(\lambda_{\text{ind}}^{-1/3})$ and smaller period $T^* = \Theta(\lambda_{\text{ind}}^{-1/3})$. These two cases represent most practical checkpointing protocols implemented in today's fault-tolerant systems. To the best of our knowledge, the results are the first to analytically establish the relationship between P^* and T^* as a function of the resource MTBF $\mu_{\text{ind}} = 1/\lambda_{\text{ind}}$.

Finally, we point out that when both checkpointing and verification costs reduce with P (which is rarely the case in practice), first-order approximation has its limitation and can no longer be used to derive the optimal number of processors and optimal checkpointing period. In this case, one can resort to higher-order approximations or numerical methods to compute the optimal pattern parameters, which are still bounded due to the sequential fraction.

6.5 Experiments

In this section, we conduct simulations to support the analytical study and to demonstrate the accuracy of first-order approximation under different parameter settings and resilience scenarios. The simulation code is publicly available for download at `http://perso.ens-lyon.fr/aurelien.cavelan/simu.zip`.

6.5.1 Simulation settings

We consider four real platforms that were used to evaluate the Scalable Checkpoint/Restart (SCR) library [74]. Measurements of the platform parameters are provided, including error rates of different sources and various checkpointing costs on a specified number of processors (where each processor has a dual quad-core chip). Following Chapter 2, the verification cost is set to be the same as that of an in-memory checkpoint, assuming the entire memory footprint needs to be inspected in order to accurately detect silent errors. Table II presents the main parameters of the four platforms. The downtime is set to one hour, i.e., $D = 3600s$ (a repair-based restoration value, see the discussion in Section 6.5.2), and the sequential fraction of the application is set to be $\alpha = 0.1$. These values as well as the individual error rate λ_{ind} will be varied in the simulations to assess their impacts on the performance of the optimal pattern.

We envision six resilience scenarios, as shown in Table III, depending on the scalability of the checkpointing and verification overheads discussed in Section 6.3. Altogether, these scenarios cover a wide range of resilience protocols, represented by different checkpointing mechanisms and error detection algorithms. For each scenario, we can compute the resilience parameters (i.e., a, b, c, v, u) based on the values of C_P and V_P as well as the number of processors given in Table II, and then project the corresponding overheads on any number of processors. The optimal pattern under each scenario can be derived using the first-order analysis presented in Section 6.4. Specifically, for a constant $\alpha > 0$, scenarios 1 and 2 correspond to case 1 $\big(C_P = cP + o(P)\big)$, scenarios 3, 4 and 5 correspond to case 2 $\big(C_P + V_P = d + o(1)\big)$, and scenario 6 corresponds to case 3 $\big(C_P + V_P = h/P\big)$. To assess the accuracy of the first-order approximation, we also compare the performance of the first-order solution with that of the optimal solution obtained using numerical methods such as the one considered in [65].

Once the pattern parameters are determined, fail-stop and silent errors are injected into the simulator as two independent Poisson processes according to the error rates shown in Table II. The result of each experiment is obtained by averaging over 500 simulation runs, each of which lasts at least 500 patterns. The expected execution overhead of the pattern is computed as the average ratio of the application's execution time with faults and its fault-free execution time.

Table II
PLATFORM PARAMETERS.

Platform	Hera	Atlas	Coastal	Coastal SSD
λ_{ind}	1.69e-8	1.62e-8	2.34e-9	2.34e-9
f	0.2188	0.0625	0.1667	0.1667
s	0.7812	0.9375	0.8333	0.8333
P	512	1024	2048	2048
C_P	$300s$	$439s$	$1051s$	$2500s$
V_P	$15.4s$	$9.1s$	$4.5s$	$180s$

Table III
DIFFERENT RESILIENCE SCENARIOS.

Scenario	1	2	3	4	5	6
C_P, R_P	cP	cP	a	a	b/P	b/P
V_P	v	u/P	v	u/P	v	u/P

6.5.2 Simulation results

Performance of optimal patterns in different scenarios

Figure 6.2 shows the performance of the optimal patterns in different resilience scenarios when the sequential fraction of the application is fixed at $\alpha = 0.1$. We can see that, on all the four platforms, the first-order solution provides a very good approximation to the optimal solution in terms of both checkpointing period and processor allocation, under the first four scenarios (the most realistic ones in practical systems). The execution overheads (≈ 0.11) predicted by the first-order formulas (Theorems 22 and 23) are almost identical to the optimal overheads and the ones obtained by simulations. The results confirm the validity of first-order approximations in these scenarios.

In scenario 5, the resilience cost is dominated by the verification overhead, which although is a constant has a relatively small value. This significantly increases the optimal processor count and hence the corresponding error rates, thus compromising the accuracy of the first-order approximation (up to 5% in execution overhead), since the lower-order terms start to become non-negligible. In fact, due to the small constant overhead, scenario 5 closely resembles scenario 6, in which case first-order analysis can no longer predict the optimal pattern parameters within the approximation limit (thus only the results of numerical methods are shown). Figure 6.2 shows that the optimal pattern parameters in scenario 6 are indeed in the same orders as those of scenario 5 but with higher processor counts and smaller checkpointing periods.

Impact of processor allocation

We study how the number of allocated processors impacts the optimal checkpointing period and the resulting execution overhead under different resilience scenarios. Figure 6.3 shows the simulation results for the Hera platform (the results are similar for the other platforms). Since the resilience overhead is dominated by the checkpointing cost, the pattern behaviors are mainly influenced by the form of C_P, as demonstrated by the almost overlapping curves between the scenarios that share the same C_P values. In all scenarios, the checkpointing period decreases with the number of processors (Figure 6.3(a)), which is needed to compensate for the increased error rates. The execution overhead, on the other hand, first improves with the number of processors due to increased parallelism and then degrades due to more errors striking (Figure 6.3(b)). The optimal processor counts, as we have seen in Figure 6.2, tend to be higher for scenarios in which the checkpointing cost C_P does not increase (or even decreases) with P. Figure 6.3(c) shows the difference in execution overhead between the first-order solution and the optimal numerical solution. The difference, for the concerned range of processors, is always within 0.2%, validating once again the accuracy of first-order approximation.

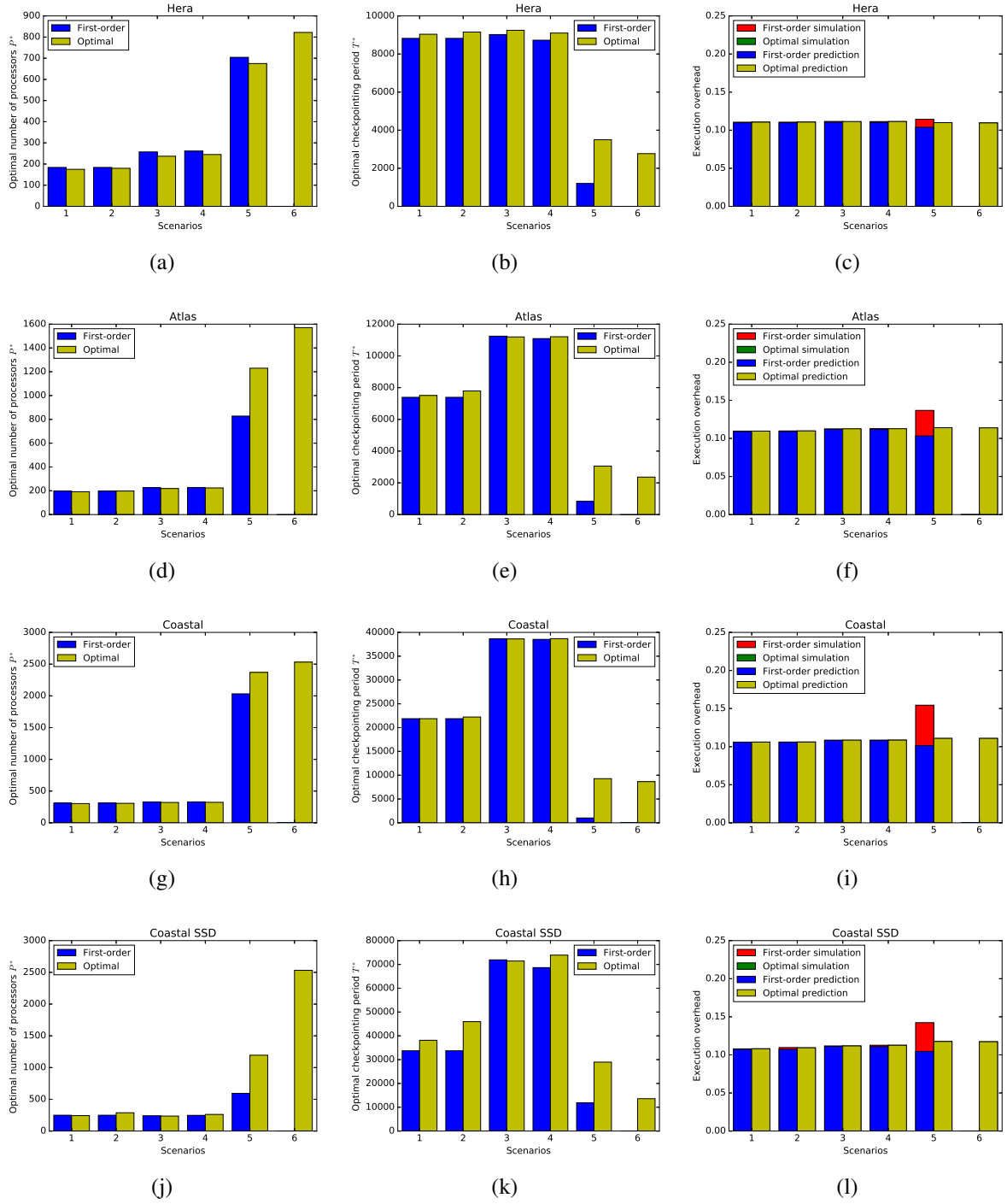
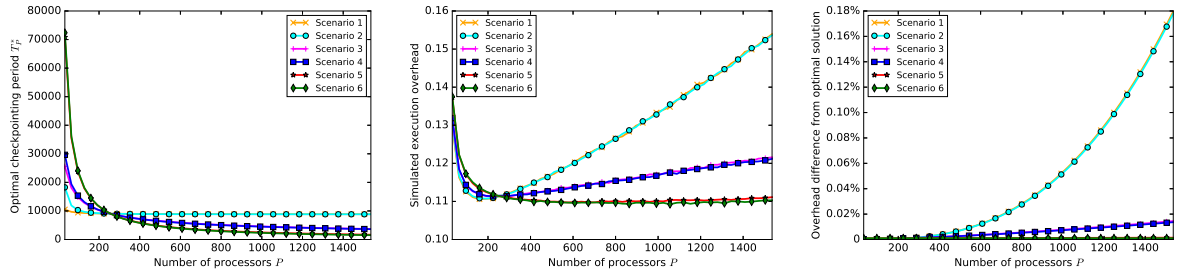
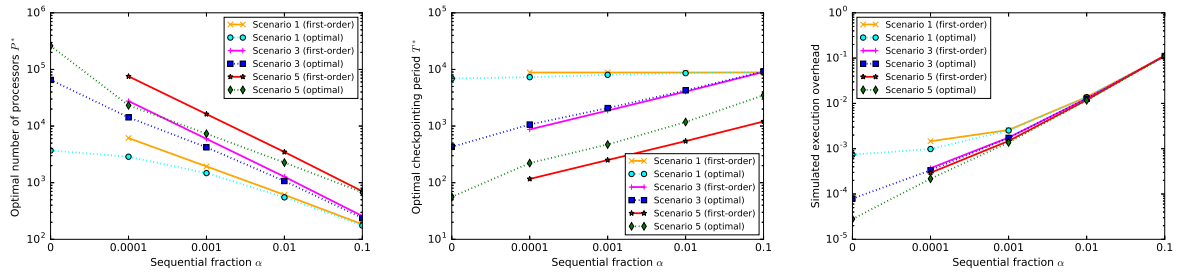


Figure 6.2: Performance of the optimal pattern in different resilience scenarios on four platforms when the sequential fraction is fixed at $\alpha = 0.1$.



(a) (b) (c)

Figure 6.3: Optimal checkpointing period T_P^* (from Theorem 21) and simulated execution overhead for different number of processors P on platform Hera.

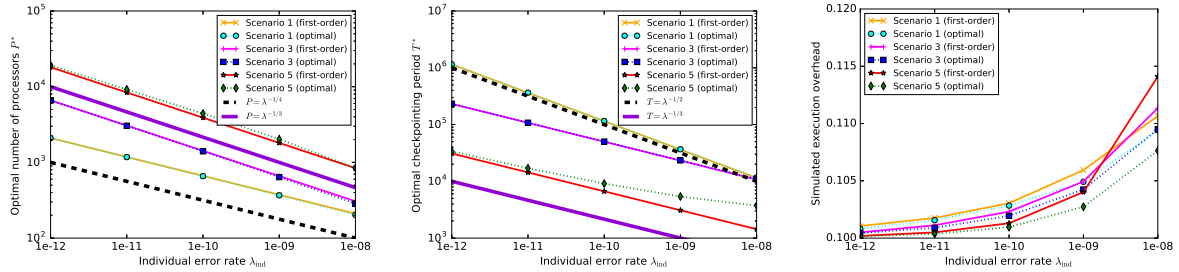


(a) (b) (c)

Figure 6.4: Optimal checkpointing period T^* and number of processors P^* (from Theorems 22 and 23, and from numerical solution), as well as the simulated execution overhead under different sequential fraction α on platform Hera.

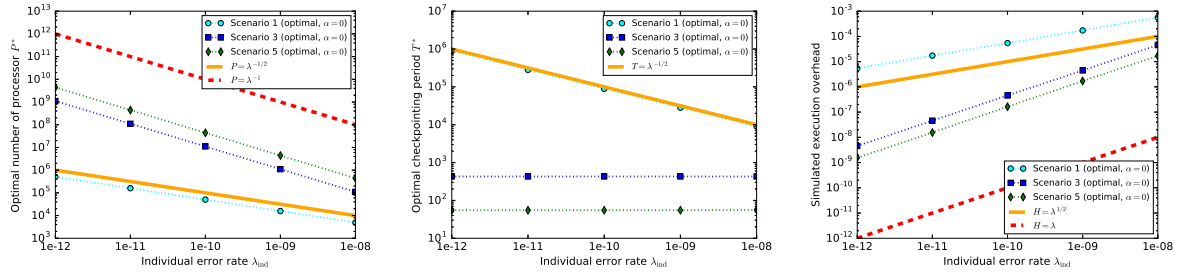
Impact of sequential fraction α

Figure 6.4 shows the impact of the sequential fraction α on the performance of the optimal patterns in scenarios 1, 3 and 5 (from now on, scenarios 2, 4 and 6 are ignored because they have similar performance as scenarios 1, 3 and 5, respectively). We can see that, as α decreases, more processors are enrolled so that the application can benefit from Amdahl's law to achieve lower execution overheads (or equivalently higher speedups). The checkpointing periods, on the other hand, decrease with α due to increased processor count, except in scenario 1 where the optimal period barely changes with the number of processors (see Theorem 21 and Figure 6.3(a)). As more processors are used, the first-order approximation of P^* starts to deviate from the optimal value, but the first-order overhead H^*, as shown in Figure 6.4(c), remains in close proximity to the optimal overhead up to $\alpha = 0.0001$. Also, compared to the other scenarios, scenario 5 starts to show a better overhead as α becomes smaller, because of its smaller checkpointing cost. However, even when $\alpha = 0$, the optimal processor allocation is upper bounded by 10^6 in all three scenarios with an overhead strictly above 10^{-5}. This is in clear contrast to the error-free execution, where an infinity number of processors can be used to achieve (nearly) null overhead.



(a) (b) (c)

Figure 6.5: Optimal checkpointing period T^* and number of processors P^* (from Theorems 22 and 23, and from numerical solution), as well as the simulated execution overhead under different values of λ_{ind} and when $\alpha = 0.1$ on platform Hera.



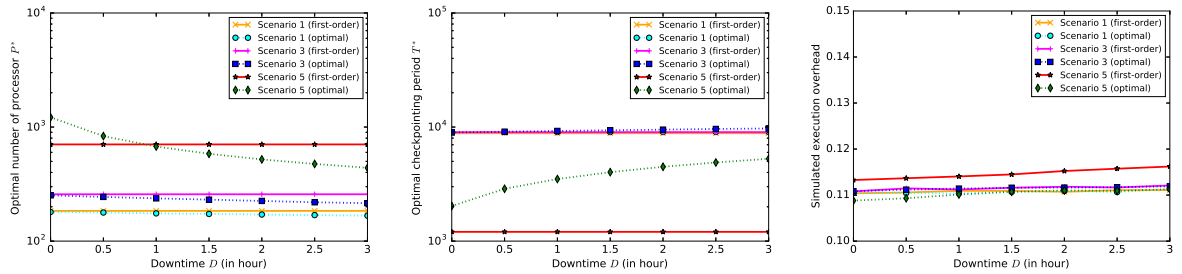
(a) (b) (c)

Figure 6.6: Optimal checkpointing period T^* and number of processors P^* (from numerical solution), as well as the simulated execution overhead under different values of λ_{ind} and when $\alpha = 0$ on platform Hera.

Impact of error rate λ_{ind}

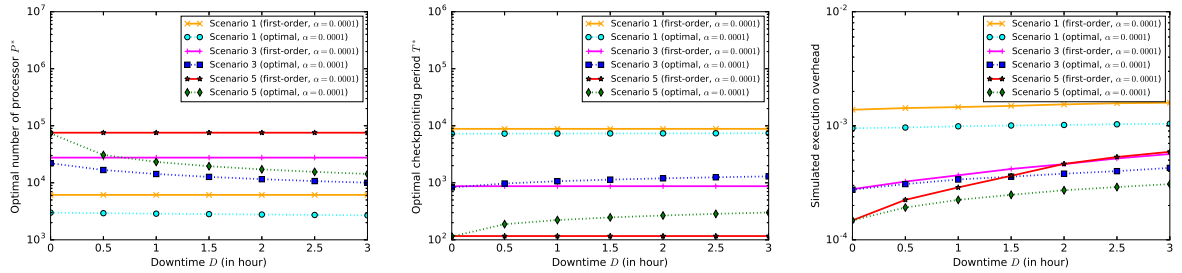
This experiment assesses the impact of the individual error rate λ_{ind} on the performance of the optimal pattern, in particular on the asymptotic behaviors of the processor allocations and checkpointing periods under different scenarios. Figure 6.5 shows that, as the processors become more reliable (i.e., as λ_{ind} decreases), the optimal pattern is able to both accommodate more processors and use larger checkpointing periods. The results confirm our analytical study that P^* and T^* are in the orders of $\lambda_{\text{ind}}^{-1/4}$ and $\lambda_{\text{ind}}^{-1/2}$, respectively, under scenario 1, and are both in the order of $\lambda_{\text{ind}}^{-1/3}$ under scenarios 3 and 5. Moreover, the first-order approximation becomes more accurate with decreased λ_{ind}, and the execution overheads tend to the theoretical lower bound of 0.1 for all three scenarios.

Figure 6.6 further shows the behaviors of the optimal patterns when the application is perfectly parallel (i.e., $\alpha = 0$). Recall that this case does not admit a solution under first-order approximation. Numerical analysis suggests that, under scenario 1, the optimal solution satisfies $P^* \approx \Theta(\lambda_{\text{ind}}^{-1/2})$, $T^* \approx \Theta(\lambda_{\text{ind}}^{-1/2})$, and $H^* \approx \Theta(\lambda_{\text{ind}}^{1/2})$, and under scenarios 3 and 5, we have $P^* \approx \Theta(\lambda_{\text{ind}}^{-1})$, $T^* \approx O(1)$, and $H^* \approx \Theta(\lambda_{\text{ind}})$.



(a) (b) (c)

Figure 6.7: Optimal checkpointing period T^* and number of processors P^* (from Theorems 22 and 23, and from numerical solution), as well as the simulated execution overhead under different downtime D and when $\alpha = 0.1$ on platform Hera.



(a) (b) (c)

Figure 6.8: Optimal checkpointing period T^* and number of processors P^* (from Theorems 22 and 23, and from numerical solution), as well as the simulated execution overhead under different downtime D and when $\alpha = 0.0001$ on platform Hera.

Impact of downtime D

Finally, we evaluate the impact of downtime D on the pattern performance. Depending on if repair-based or replacement-based (migration to a spare processor) restoration is used, downtime can range from a few minutes to several hours [65]. In this experiment, we vary the downtime from 0 to 3 hours. Figures 6.7 and 6.8 show the simulation results when $\alpha = 0.1$ and $\alpha = 0.0001$, respectively. Since D does not appear in the formulas of P^* and T^* (given in Theorems 22 and 23) due to the use of first-order approximation, the optimal pattern obtained by the first-order analysis does not vary with D, while the optimal processor count obtained by the numerical solution decreases with increased downtime. This shows that the optimal pattern parameters are indeed influenced by the downtime. However, the simulated execution overheads in both cases stay close for the first-order solution and the optimal solution, because even a 3-hour downtime is nevertheless much smaller compared to the platform MTBF (in the order of days).

6.6 Conclusion

In this chapter, we considered the optimal processor allocation problem for executing a parallel job on a large-scale platform subject to fail-stop and silent errors. We have provided the exact expression for the expected execution time of a pattern, and closed-form first-order approximation formulas to compute the optimal checkpointing period T^* and optimal number of processors P^*. These formulas are functions of several parameters: the individual processor failure rate λ_{ind}, the sequential fraction of the application α, as well as the checkpointing and verification costs C_P and V_P. For the latter costs, we have envisioned a comprehensive set of scenarios that are representative of the most important fault-tolerant protocols. To the best of our knowledge, these results are the first that analytically establish the relationship between P^* and T^* as a function of the resource MTBF $\mu_{\text{ind}} = 1/\lambda_{\text{ind}}$, and they offer new insights into the relationships of Amdhal's law and the Young/Daly approximation formula. Also, they provide the first (and direct) characterization of the optimal number of resources to enroll, with given error rates, resilience costs and application speedup profile. We have conducted an extensive set of simulations to support the theoretical study, whose outcome confirms the accuracy of first-order approximation under a wide range of parameter settings.

Chapter 7

Identifying the Right Replication Level for Detecting and Correcting Silent Errors

This chapter provides a model and an analytical study of replication as a technique to detect and correct silent errors. Although other detection techniques exist for HPC applications, based on algorithms (ABFT), invariant preservation or data analytics, replication remains the most transparent and least intrusive technique. We explore the right level (duplication, triplication or more) of replication needed to efficiently detect and correct silent errors. Replication is combined with checkpointing and comes with two flavors: *process replication* and *group replication*. Process replication applies to message-passing applications with communicating processes. Each process is replicated, and the platform is composed of process pairs, or triplets. Group replication applies to black-box applications, whose parallel execution is replicated several times. The platform is partitioned into two halves (or three thirds). In both scenarios, results are compared before each checkpoint, which is taken only when both results (duplication) or two out of three results (triplication) coincide. If not, one or more silent errors have been detected, and the application rolls back to the last checkpoint. We provide a detailed analytical study of both scenarios, with formulas to decide, for each scenario, the optimal parameters as a function of the error rate, checkpoint cost, and platform size. We also report a set of extensive simulation results that corroborates the analytical model. This work has been accepted to the *Fault Tolerance for HPC at eXtreme Scale* (FTXS'2017) workshop [W1].

7.1 Introduction

Triple Modular Redundancy, or TMR [72], is the standard fault-tolerance approach for critical systems, such as embedded or aeronautical devices [5]. With TMR, computations are executed three times, and a majority voting is conducted to select the correct result out of the three available ones. Indeed, if two or more results agree, they are declared correct, because the probability of two or more errors leading to the same wrong result is assumed so low that it can be ignored. While triplication seems very expensive in terms or resources, anybody sitting in a

plane would heartily agree that it is worth the price.

On the contrary, duplication, let alone triplication, has a bad reputation in the High Performance Computing (HPC) community. Who would be ready to waste half or two-thirds of precious computing resources? However, despite its high cost, several authors have been advocating the use of duplication in HPC in the recent years [49, 52, 86, 100]. In a nutshell, this is because platform sizes have become so large that fail-stop errors are likely to strike at a high rate during application execution. 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 [62]). 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. Given any value of μ_{ind}, there is a threshold value for the number of processors above which platform throughput will decrease [47, 52, 77, 86]: the platform MTBF becomes so small that the applications experience too many failures, hence too many recoveries and re-execution delays, to progress efficiently. All this explains why duplication has been considered for HPC applications despite its cost. The authors in [52] propose *process replication* by which each process in a parallel MPI (Message Passing Interface) application is duplicated on multiple physical processors while maintaining synchronous execution of the replicas. This approach is effective because the MTBF of a set of two replicas (which is the average delay for failures to strike *both* processors in the replica set) is much larger than the MTBF of a single processor.

Process replication may not always be a feasible option. Process replication features must be provided by the application. Some prototype MPI implementations [52, 53] are convincing proofs of concept and do provide such capabilities. However, many other programming frameworks (not only MPI-like frameworks, but also concurrent objects, distributed components, workflows, algorithmic skeletons) do not provide an equivalent to transparent process replication for the purpose of fault-tolerance, and enhancing them with transparent replication may be non-trivial. When transparent replication is not (yet) provided by the runtime system, one solution could be to implement it explicitly within the application, but this is a labor-intensive process especially for legacy applications. Another approach introduced in [26] is *group replication*, a technique that can be used whenever process replication is not available. Group replication is agnostic to the parallel programming model, and thus views the application as an unmodified black box. The only requirement is that the application be startable from a saved checkpoint file. Group replication consists in executing multiple application instances concurrently. For example, 2 distinct P-process application instances could be executed on a $2P$-processor platform. At first glance, it may seem paradoxical that better performance can be achieved by using group duplication. After all, in the above example, 50% of the platform is "wasted" to perform redundant computation. The key point here is that each application instance runs at a smaller scale. As a result each instance can use lower checkpointing frequency, and can thus have better parallel efficiency in the presence of faults, when compared to a single application instance running at full scale. In some cases, the application makespan can then be comparable to, or even shorter than that obtained when running a single application instance. In the end, the cost of wasting processor power for redundant computation can be offset by the benefit of reduced checkpointing frequency. Furthermore, in group replication, once an instance saves a checkpoint, the other instance can use this checkpoint immediately

to "jump ahead" in its execution. Hence, group replication is more efficient than the mere independent execution of several instances: each time one instance successfully completes a given "chunk of work", all the other instances immediately benefit from this success. To implement group replication the runtime system needs to perform the typical operations needed for system-assisted checkpoint/restart: determining checkpointing frequencies for each application instance, causing checkpoints to be saved, detecting application failures, and restarting an application instance from a saved checkpoint after a failure. The only additional feature is that the system must be able to stop an instance and cause it to resume execution from a checkpoint file produced by another instance as soon as it is produced.

Process or group replication has been mainly proposed in HPC to cope with fail-stop errors. However, another challenge is represented by silent errors, or silent data corruption, whose threat can no longer be ignored [74, 76, 102]. There are several causes of silent errors, such as cosmic radiation, packaging pollution, among others. Silent errors can strike the cache and memory (bit flips) as well as CPU operations; in the latter case they resemble floating-point errors due to improper rounding, but have a dramatically larger impact because any bit of the result, not only low-order mantissa bits, can be corrupted. In contrast to a fail-stop error whose detection is immediate, a silent error is identified only when the corrupted data leads to an unusual application behavior. Such detection latency raises a new challenge: if the error struck before the last checkpoint, and is detected after that checkpoint, then the checkpoint is corrupted and cannot be used for rollback. To distinguish from fail-stop failures, we use MTBE instead of MTBF to characterize the rate of silent errors.

To address the problem of silent errors, many application-specific detectors, or verification mechanisms, have been proposed (see Section 7.2 for a survey). It is not clear, however, whether a special-purpose detector can be designed for each scientific application. In addition, application-specific verification mechanisms only protect from some types of error sources, and fail to provide accurate and efficient detection of all silent errors. In fact, providing such detectors for scientific applications has been identified as one of the hardest challenges[1] towards extreme-scale computing [24, 25].

Altogether, silent errors call for revisiting replication in the framework of scientific application executing on large-scale HPC platforms. Because replication is now applied at the process level, scale becomes an even harder-to-fight enemy. Processor count ranges to about 10^5 on the K-computer and TaihuLight systems. The number of processors could increase further to 10^6 (hence 10^6 or more processes) on Exascale systems, with billions of threads [40]. In addition, the probability of several errors striking during an execution can get significant, depending upon whether or not circuit manufacturers increase significantly the protection of the logic, latch/flip-flops and static arrays in the processor. In a recent paper [89], the authors consider that with significant more protection (more hardware, more power consumption), the FIT[2] rate for undetected errors on a processor circuit could be maintained to around 20. But without additional protection compared to the current situation, the FIT rate for undetected errors could be as high as 5,000 (or 1 error every 200,000 hours). Combining 10 million of devices with

[1] More generally, trustworthy computing, which aims at guaranteeing the correctness of the results of a long-lasting computation on a large-scale supercomputer, has received considerable attention recently [23].

[2] The Failures in Time (FIT) rate of a device is the number of failures that can be expected in one billion (10^9) device-hours of operation.

this FIT rate would result in a silent error in the system every 72 seconds. This work aims at providing a quantitative assessment of the potential of duplication and triplication to mitigate such a threat. Specifically, the main contributions of this work are:

- an analytical model to study the performance of all replication scenarios against silent errors, namely, duplication, triplication, or more for process and group replications;
- closed-form formulas that give the optimal checkpointing period and optimal process number as a function of the error rate, checkpoint cost, and platform size;
- a set of simulation results that corroborate the analytical model.

The rest of the chapter is organized as follows. Section 7.2 surveys the related work. We introduce the performance model in Section 7.3, and derive the general expected execution time in Section 7.4. The analysis for process replication is presented in Section 7.5, followed by the analysis for group replication in Section 7.6. Section 7.7 is devoted to the simulation results. Finally, we provide concluding remarks and directions for future work in Section 7.8.

7.2 Related work

We survey related work in this section. We start with replication for HPC applications in Section 7.2.1 and cover application-specific detectors in Section 7.2.2.

7.2.1 Replication for fail-stop errors

Checkpointing policies have been widely studied. We refer to [62] for a survey of various protocols and the derivation of the Young's and Daly's formula [36, 97] for the optimal checkpointing periods. Recent advances include multi-level approaches, or the use of SSD or NVRAM as secondary storage [25]. Combining replication with checkpointing has been proposed in [49, 86, 100] for HPC platforms, and in [69, 96] for grid computing.

The use of redundant MPI processes is analyzed in [27, 50, 52]. In particular, the work by Ferreira et al. [52] has studied the use of process replication for MPI applications, using 2 replicas per MPI process. They provide a theoretical analysis of parallel efficiency, an MPI implementation that supports transparent process replication (including failure detection, consistent message ordering among replicas, etc.), and a set of experimental and simulation results. Partial redundancy is studied in [45, 91] (in combination with coordinated checkpointing) to decrease the overhead of full replication. Adaptive redundancy is introduced in [59], where a subset of processes is dynamically selected for replication.

Thread-level replication has been investigated in [35, 82, 98]. This work targets process-level replication, in order to be able to detect (and correct) silent errors striking in all communication-related operations.

Finally, Ni et al [75] introduce process duplication to cope with both fail-stop and silent errors. Their pioneering paper contains many interesting results but differs from this work as follows: (i) they limit themselves to perfectly parallel applications while we investigate

speedup profiles that obey Amdahl's law; (ii) they do not investigate triplication; and (iii) they compute an upper bound on the optimal period and do not determine optimal processor counts.

7.2.2 Silent error detection and correction

Application-specific information enables ad-hoc solutions, which dramatically decrease the cost of error detection. Algorithm-based fault tolerance (ABFT) [16, 63, 87] is a well-known technique, which uses checksums to detect up to a certain number of errors in linear algebra kernels. Unfortunately, ABFT can only protect datasets in linear algebra kernels, and it must be implemented for each different kernel, which incurs a large amount of work for large HPC applications. Other techniques have also been advocated. Benson, Schmit and Schreiber [14] compare the result of a higher-order scheme with that of a lower-order one to detect errors in the numerical analysis of ODEs and PDEs. Sao and Vuduc [85] investigate self-stabilizing corrections after error detection in the conjugate gradient method. Bridges et al. [19] propose linear solvers to tolerant soft faults using selective reliability. Elliot et al. [44] design a fault-tolerant GMRES capable of converging despite silent errors. Bronevetsky and de Supinski [21] provide a comparative study of detection costs for iterative methods.

Recently, several silent error detectors based on data analytics have been proposed, showing promising results. These detectors use several interpolation techniques such as time series prediction [15] and spatial multivariate interpolation [8, 9, 11]. Such techniques offer large detection coverage for a negligible overhead. However, these detectors do not guarantee full coverage; they can detect only a certain percentage of corruptions (i.e., partial verification with an imperfect recall). Nonetheless, the accuracy-to-cost ratios of these detectors are high, which makes them interesting alternatives at large scale. Similar detectors have also been designed to detect silent errors in the temperature data of the Orbital Thermal Imaging Spectrometer (OTIS) [31].

Again, all the papers quoted in this section provide application-specific detectors, while our approach is agnostic of the application characteristics. The only information is whether we can use either process replication. If not, we see the application as a black box and can use only group replication.

7.3 Model

This section presents the analytical model for evaluating the performance of different replication scenarios. The model is classical, similar to those of the literature for replication [52], only with a different objective (quantifying replication for silent errors). Table I summarizes the main notations.

Recall that μ_{ind} denotes the MTBE of an individual processor or process[3] of the system, and let $\lambda = \frac{1}{\mu_{\text{ind}}}$ denote the silent error rate of the processor. The error rate for a collection of P processors is then given by $\lambda_P = \frac{1}{\mu_P} = \frac{P}{\mu_{\text{ind}}} = \lambda P$ [62]. Assuming that the error arrivals follow

[3]We assume that each process is executed by a dedicated processor, hence use "processor" and "process" interchangeably. We also use MTBE instead of MTBF to emphasize that we deal with (silent) errors, not failures.

Table I
LIST OF NOTATIONS.

Parameters	
T	Length (or period) of a pattern
P	Number of processes allocated to an application
n	Number of (process or group) replicas
$S(P)$	Speedup function of an application
$H(P) = \frac{1}{S(P)}$	Error-free execution overhead
$\mathbb{E}_n(T, P)$	Expected execution time of a pattern
$\mathbb{H}_n(T, P)$	Expected execution overhead of a pattern
$\mathbb{S}_n(T, P)$	Expected speedup function of a pattern
$\lambda = \frac{1}{\mu_{\text{ind}}}$	Silent error rate of an individual process
$\mathbb{P}_n(T, P)$	Silent error probability of a pattern
C	Checkpointing cost
R	Recovery cost
V	Verification cost (comparison of replicas)

Exponential distribution, the probability that a computation hit by a silent error during time T on P processes is given by $\mathbb{P}(T, P) = 1 - e^{-\lambda PT}$.

Consider long-latsting HPC applications that execute for hours or even days on a large-scale platform. Resilience is enforced by the combined use of replication and periodic checkpointing. Before each checkpoint, the results of different replicas are compared. Only when both results (for duplication) or two out of three results (for triplication) coincide[4], in which case a *consensus* is said to be reached, the checkpoint is taken. Otherwise, silent errors are assumed to have been detected, and they cannot be corrected through consensus. The application then rolls back to the last checkpoint. There are two different types of replications:

- *Process replication*: Each process of the application is replicated, and the results of different processes are independently compared. A rollback is needed when at least one process has failed to reach a consensus;

- *Group replication*: The entire application (as a black box) is replicated, and the results of all replicas (as a whole) are compared. A rollback is needed when these group replicas fail to reach a consensus.

The computational chunk between two checkpoints is called a *periodic pattern*. For a replication scenario with n replicas, the objective is to minimize the expected total execution time (or makespan) of an application by finding the optimal pattern parameters:

- T: length (or period) of the pattern;

- P: number of processes allocated to the application.

[4]For $n > 3$ replicas, the results of k replicas should coincide, where $2 \leq k < n$ is a design parameter set by the system to control the level of reliability. $k = \lfloor \frac{n}{2} \rfloor + 1$ is a widely-used choice (majority voting).

Indeed, for long-lasting applications, it suffices to focus on just one pattern, since the pattern repeats itself over time. To see this, let W_{total} denote the total amount of work of the application and suppose the application has a speedup function $S(P)$ when executed on P processors. In this chapter, we focus on a speedup function that obeys Amdahl's law[5]:

$$S(P) = \frac{1}{\alpha + \frac{1-\alpha}{P}}\ , \tag{7.1}$$

where $\alpha \in [0, 1]$ denotes the sequential fraction of the application that cannot be parallelized. For convenience, we also define $H(P) = \frac{1}{S(P)}$ to be the execution overhead. For a pattern of length T and run by P processes, the amount of work done in a pattern is therefore $W_{\text{pattern}} = T \cdot S(P)$, and the total number of patterns in the application can be approximated as $m = \frac{W_{\text{total}}}{W_{\text{pattern}}} = \frac{W_{\text{total}}}{T \cdot S(P)} = \frac{W_{\text{total}}}{T} H(P)$. Now, let $\mathbb{E}_n(T, P)$ denote the expected execution time of the pattern with n replicas in either replication scenario. Define $\mathbb{H}_n(T, P) = \frac{\mathbb{E}_n(T,P)}{T} H(P)$ to be the expected execution overhead of the pattern, and $\mathbb{S}_n(T, P) = \frac{1}{\mathbb{H}_n(T,P)}$ the expected speedup. The expected makespan of the application can then be written as $\mathbb{E}_{\text{total}} \approx \mathbb{E}_n(T, P)m = \mathbb{E}_n(T, P)\frac{W_{\text{total}}}{T} H(P) = \mathbb{H}_n(T, P) \cdot W_{\text{total}} = \frac{W_{\text{total}}}{\mathbb{S}_n(T,P)}$. This shows that the optimal expected makespan can be achieved by minimizing the expected execution overhead of a pattern, or equivalently, maximizing the expected speedup.

Now, we describe a model for the costs of checkpoint, recovery and consensus verification. First, the checkpoint cost clearly depends on the protocol and storage type. Note that only the result of one replica needs to be checkpointed, so the cost does not increase with the number of replicas. To save the application's memory footprint M to the storage system using P processes, we envision the following two scenarios:

- $C = \frac{M}{\tau_{io}}$: In this case, checkpoints are being written to the remote storage system, whose bandwidth is the I/O bottleneck. Here, τ_{io} is the remote I/O bandwidth.
- $C = \frac{M}{\tau_{net} P}$: This case corresponds to in-memory checkpoints, where each process stores $\frac{M}{P}$ data locally (e.g., on SSDs). Here, τ_{net} is the process network bandwidth.

The recovery cost is assumed to be the same as the checkpointing cost, i.e., $R = C$, as it involves the same I/O operations. This is a common assumption [74], although practical recovery cost can be somewhat smaller than the checkpoint cost [37]. Finally, verifying consensus is performed by communicating and comparing $\frac{M}{P}$ data stored on each process, which can be executed concurrently by all process pairs (or triplets). Hence, the verification cost satisfies $V = O(\frac{M}{P})$. Overall, we use the following general expression to account for the combined cost of verification and checkpoint/recovery:

$$V + C = c + \frac{d}{P}\ , \tag{7.2}$$

where c and d are constants that depend on the application memory footprint, checkpointing protocol, network or I/O bandwidth, etc. Equation (7.2) is convenient in terms of analysis as we will see in the subsequent sections. Here, $c = 0$ corresponds to the second checkpointing scenario discussed above.

[5]The model is generally applicable to other speedup functions as well.

7.4 Expected execution time

In this section, we compute the expected execution time of a periodic pattern, which will be used in the next two sections to derive the optimal pattern parameters.

Theorem 24. *The expected time to execute a periodic pattern of length T using P processes and n replicas can be expressed as*

$$\mathbb{E}_n(T,P) = T + V + C + \frac{\mathbb{P}_n(T,P)}{1-\mathbb{P}_n(T,P)}\,(T+V+R)\,, \tag{7.3}$$

where $\mathbb{P}_n(T,P)$ denotes the probability that the execution fails due to silent errors striking during the pattern and we have to roll back to the last checkpoint.

Proof. Since replicas are synchronized, we can generally express the expected execution time as follows:

$$\mathbb{E}_n(T,P) = T + V + \mathbb{P}_n(T,P)\big(R + \mathbb{E}_n(T,P)\big) + \big(1-\mathbb{P}_n(T,P)\big)C\,. \tag{7.4}$$

First, the pattern of length T is executed followed by the verification (through comparison and/or voting), which incurs cost V. With probability $\mathbb{P}_n(T,P)$, the pattern fails due to silent errors. In this case, we need to re-execute the pattern after performing a recovery from the last checkpoint with cost R. Otherwise, with probability $1-\mathbb{P}_n(T,P)$, the execution succeeds and the checkpoint with cost C is taken at the end of the pattern. Now, solving for $\mathbb{E}_n(T,P)$ from Equation (7.4), we can obtain the expected execution time of the pattern as shown in Equation (7.3). □

Remarks. Theorem 24 is applicable to both process replication and group replications. The only difference lies in the computation of failure probability $\mathbb{P}_n(T,P)$, which depends not only on the replication scenario but also on the number of replicas n.

7.5 Process replication

In this section, we consider process replication. We first derive the optimal computing patterns when each process of the application is duplicated (Section 7.5.1) and triplicated (Section 7.5.2), respectively. Finally, we generalize the results to an arbitrary but constant number of replications per process under a general process replication framework (Section 7.5.3).

7.5.1 Process duplication

We start with process duplication, that is, each process has two replicas. The following lemma shows the failure probability of a given computing pattern in this case.

Lemma 12. *Using process duplication, the failure probability of a computing pattern of length T and with P processes is given by*

$$\mathbb{P}_2^{\text{prc}}(T,P) = 1 - e^{-2\lambda TP}\,. \tag{7.5}$$

Proof. With duplication, errors cannot be corrected (no consensus), hence a process fails if either one of its replicas fails or both replicas fail. In other words, there is an error if the results of both replicas do not coincide (we neglect the quite unlikely scenario with one error in each replica leading to the same wrong result). Let $\mathbb{P}_1^{\text{prc}}(T,1) = 1 - e^{-\lambda T}$ denote the probability of a single process failure. Therefore, we can write the failure probability of any duplicated process as follows:

$$\begin{aligned}\mathbb{P}_2^{\text{prc}}(T,1) &= \binom{2}{1}\left(1-\mathbb{P}_1^{\text{prc}}(T,1)\right)\mathbb{P}_1^{\text{prc}}(T,1) + \mathbb{P}_1^{\text{prc}}(T,1)^2 \\ &= 2e^{-\lambda T}\left(1-e^{-\lambda T}\right) + \left(1-e^{-\lambda T}\right)^2 \\ &= 1 - e^{-2\lambda T}\,.\end{aligned}$$

Now, because we have P independent processes, the probability that the application gets interrupted by silent errors is the probability that at least one process fails because of silent errors, which can be expressed as:

$$\begin{aligned}\mathbb{P}_2^{\text{prc}}(T,P) &= 1 - \mathbb{P}(\text{“No process fails”}) \\ &= 1 - \left(1-\mathbb{P}_2^{\text{prc}}(T,1)\right)^P \\ &= 1 - e^{-2\lambda PT}\,.\end{aligned}$$

□

Using the failure probability in Lemma 12, we derive the optimal computing pattern for process duplication as shown in the following theorem. Recall that the application speedup follows Amdahl's law as shown in Equation (7.1) and the cost of verification and checkpoint is modeled by Equation (7.2).

Theorem 25. *A first-order approximation to the optimal number of processes for an application with 2 replicas per process is given by*

$$P_{\text{opt}} = \min\left\{\frac{Q}{2}, \left(\frac{1}{2}\left(\frac{1-\alpha}{\alpha}\right)^2\frac{1}{c\lambda}\right)^{\frac{1}{3}}\right\}\,, \tag{7.6}$$

where Q denotes the total number of available processes in the system. The associated optimal checkpointing period and the expected speedup function of the application are

$$T_{\text{opt}}(P_{\text{opt}}) = \left(\frac{V+C}{2\lambda P_{\text{opt}}}\right)^{\frac{1}{2}}\,, \tag{7.7}$$

$$\mathbb{S}_2^{\text{prc}}(P_{\text{opt}}) = \frac{S(P_{\text{opt}})}{1+2\left(2\lambda(V+C)P_{\text{opt}}\right)^{\frac{1}{2}}}\,. \tag{7.8}$$

Proof. First, we can derive, from Theorem 24 and Lemma 12, the expected execution time of a pattern with length T and P duplicated processes as follows:

$$\begin{aligned}\mathbb{E}_2^{\text{prc}}(T,P) &= T + V + C + \left(e^{2\lambda PT} - 1\right)(T+V+R) \\ &= T + V + C + 2\lambda PT\,(T+V+R) + o(\lambda PT^2)\,.\end{aligned}$$

The second equation above is obtained by applying Taylor series to approximate $e^z = 1 + z + o(z)$ for $z < 1$, while assuming $\lambda PT = \Theta(\lambda^\epsilon)$, where $\epsilon > 0$.

Now, we have a closed-form expression for $\mathbb{E}_2^{\text{prc}}(T, P)$. Substituting it into $\mathbb{H}_2^{\text{prc}}(T, P) = H(P)\frac{\mathbb{E}_2^{\text{prc}}(T,P)}{T}$, we can get the expected execution overhead as:

$$\mathbb{H}_2^{\text{prc}}(T, P) = H(P)\left(1 + \frac{V + C}{T} + 2\lambda PT + o(\lambda PT)\right) . \tag{7.9}$$

The optimal overhead can then be achieved by balancing (or equating) the two terms $\frac{V+C}{T}$ and $2\lambda PT$ above, which gives the following optimal checkpointing period as a function of the process count:

$$T_{\text{opt}}(P) = \left(\frac{V + C}{2\lambda P}\right)^{\frac{1}{2}} . \tag{7.10}$$

Now, substituting $T_{\text{opt}}(P)$ back into Equation (7.9), we get the execution overhead as a function of the process count as follows (lower-order terms ignored):

$$\mathbb{H}_2^{\text{prc}}(P) = H(P)\left(1 + 2\big(2\lambda(V + C)P\big)^{\frac{1}{2}}\right) . \tag{7.11}$$

Note that Equations (7.10) and (7.11) hold true regardless of the form of the function $H(P)$ or the cost $V + C$. Recall that we consider Amhdal's law $H(P) = \alpha + \frac{1-\alpha}{P}$ and a cost model $V + C = c + \frac{d}{P}$. In order to derive the optimal process count, we consider two cases:

Case (1). $c > 0$ and $\alpha > 0$ are both constants: we can expand Equation (7.11) to be

$$\mathbb{H}_2^{\text{prc}}(P) = \alpha + 2\alpha\big(2\lambda cP\big)^{\frac{1}{2}} + \frac{1 - \alpha}{P} + o(\lambda^{\frac{1}{2}}) . \tag{7.12}$$

The optimal overhead can then be achieved by setting

$$\frac{\partial \mathbb{H}_2^{\text{prc}}(P)}{\partial P} = \alpha\left(\frac{2\lambda c}{P}\right)^{\frac{1}{2}} - \frac{1 - \alpha}{P^2} = 0 ,$$

which leads to $P^* = \left(\frac{1}{2}\left(\frac{1-\alpha}{\alpha}\right)^2 \frac{1}{c\lambda}\right)^{\frac{1}{3}}$. Since the total number of processes in the system is Q and each application process is duplicated, the optimal process count is upper-bounded by $\frac{Q}{2}$ if $P^* > \frac{Q}{2}$, due to the convexity of $\mathbb{H}_2^{\text{prc}}(P)$ as shown in Equation (7.11). Hence, the optimal process count P_{opt} is given by Equation (7.6).

Case (2). $c = 0$ or $\alpha = 0$: In either case, we can see that Equation (7.11) becomes a decreasing function of P. Therefore, the optimal strategy is to utilize all the available Q processes, i.e., $P_{\text{opt}} = \frac{Q}{2}$, which again satisfies Equation (7.6), since $\left(\frac{1}{2}\left(\frac{1-\alpha}{\alpha}\right)^2 \frac{1}{c\lambda}\right)^{\frac{1}{3}} = \infty$.

In either case, the expected application speedup is then given by the reciprocal of the overhead as shown in Equation (7.11) with the optimal process count P_{opt}. □

Remarks. For fully parallelizable applications, i.e., $\alpha = 0$, the optimal pattern on a Q-process platform is characterized by

$$P_{\text{opt}} = \frac{Q}{2}\,, \quad T_{\text{opt}} = \begin{cases} \sqrt{\frac{c}{\lambda Q}} & \text{for } V + C = c \\ \frac{1}{Q}\sqrt{\frac{2d}{\lambda}} & \text{for } V + C = \frac{d}{P} \end{cases}\,,$$

$$\mathbb{S}_2^{\text{prc}}(P_{\text{opt}}) = \begin{cases} \frac{Q}{2(1+2\sqrt{\lambda c Q})} & \text{for } V + C = c \\ \frac{Q}{2(1+2\sqrt{2\lambda d})} & \text{for } V + C = \frac{d}{P} \end{cases}\,.$$

7.5.2 Process triplication

Now, we consider process duplication, that is, each process has three replicas. This is the smallest number of replicas that allows an application to recover from silent errors through majority voting instead of rolling back to the last checkpoint.

Lemma 13. *Using process triplication, the failure probability of a computing pattern of length T and with P processes is given by*

$$\mathbb{P}_3^{\text{prc}}(T, P) = 1 - \left(3e^{-2\lambda T} - 2e^{-3\lambda T}\right)^P\,. \tag{7.13}$$

Proof. Using triplication, if only one replica fails, the silent error can be masked by the two successful replicas. Hence, in this case, a process fails if at least two of its replicas are hit by silent errors. Let $\mathbb{P}_1^{\text{prc}}(T, 1) = 1 - e^{-\lambda T}$ denote the probability of a single process failure. Therefore, we can write the failure probability of any triplicated process as follows:

$$\begin{aligned} \mathbb{P}_3^{\text{prc}}(T, 1) &= \binom{3}{2} \left(1 - \mathbb{P}_1^{\text{prc}}(T, 1)\right) \mathbb{P}_1^{\text{prc}}(T, 1)^2 + \mathbb{P}_1^{\text{prc}}(T, 1)^3 \\ &= 3e^{-\lambda T}\left(1 - e^{-\lambda T}\right)^2 + \left(1 - e^{-\lambda T}\right)^3 \\ &= 1 - 3e^{-2\lambda T} + 2e^{-3\lambda T}\,. \end{aligned}$$

For P independent processes, the application fails when at least one of its processes fails. Hence, we have:

$$\begin{aligned} \mathbb{P}_3^{\text{prc}}(T, P) &= 1 - \mathbb{P}(\text{“No process fails”}) \\ &= 1 - \left(1 - \mathbb{P}_3^{\text{prc}}(T, 1)\right)^P \\ &= 1 - \left(3e^{-2\lambda T} - 2e^{-3\lambda T}\right)^P\,. \end{aligned}$$

□

The following theorem derives the optimal computing pattern for process triplication.

Theorem 26. *A first-order approximation to the optimal number of processes for an application with 3 replicas per process is given by*

$$P_{\text{opt}} = \min\left\{\frac{Q}{3}, \left(\frac{4}{3}\left(\frac{1-\alpha}{\alpha}\right)^3 \left(\frac{1}{c\lambda}\right)^2\right)^{\frac{1}{4}}\right\}\,, \tag{7.14}$$

where Q denotes the total number of available processes in the system. The associated optimal checkpointing period and the expected speedup function of the application are

$$T_{\text{opt}}(P_{\text{opt}}) = \left(\frac{V+C}{6\lambda^2 P_{\text{opt}}}\right)^{\frac{1}{3}} , \tag{7.15}$$

$$\mathbb{S}_3^{\text{prc}}(P_{\text{opt}}) = \frac{S(P_{\text{opt}})}{1+3\left(\frac{3}{4}\left(\lambda(V+C)\right)^2 P_{\text{opt}}\right)^{\frac{1}{3}}} . \tag{7.16}$$

Proof. From Theorem 24 and Lemma 13, and applying Taylor series, we can derive the expected execution time of a pattern as follows:

$$\begin{aligned}
\mathbb{E}_3^{\text{prc}}(T,P) &= T+V+C+\frac{1-\left(3e^{-2\lambda T}+2e^{-3\lambda T}\right)^P}{\left(3e^{-2\lambda T}-2e^{-3\lambda T}\right)^P}\left(T+V+R\right) \\
&= T+V+C+\left(\left(\frac{e^{3\lambda T}}{3e^{\lambda T}-2}\right)^P-1\right)\left(T+V+R\right) \\
&\approx T+V+C+\left(\left(\frac{1+3\lambda T+\frac{(3\lambda T)^2}{2}}{1+3\lambda T+\frac{3(\lambda T)^2}{2}}\right)^P-1\right)\left(T+V+R\right) \\
&\approx T+V+C+\left(\left(1+3(\lambda T)^2\right)^P-1\right)\left(T+V+R\right) \\
&= T+V+C+\left(\sum_{j=0}^{P}\binom{P}{j}\left(3(\lambda T)^2\right)^j-1\right)\left(T+V+R\right) \\
&= T+V+C+3P(\lambda T)^2(T+V+R)+o(\lambda^2 PT^3) .
\end{aligned}$$

The execution overhead can then be expressed as:

$$\mathbb{H}_3^{\text{prc}}(T,P) = H(P)\left(1+\frac{V+C}{T}+3P(\lambda T)^2+o(\lambda^2 PT^2)\right) . \tag{7.17}$$

The optimal checkpointing period is then obtained by setting

$$\frac{\partial \mathbb{H}_3^{\text{prc}}(T,P)}{\partial T} = -\frac{V+C}{T^2}+6\lambda^2 PT = 0 ,$$

which gives

$$T_{\text{opt}}(P) = \left(\frac{V+C}{6\lambda^2 P}\right)^{\frac{1}{3}} .$$

Substituting $T_{\text{opt}}(P)$ back into Equation (7.17), we get the following execution overhead (with lower-order terms ignored):

$$\mathbb{H}_3^{\text{prc}}(P) = H(P)\left(1+3\left(\frac{3}{4}\left(\lambda(V+C)\right)^2 P\right)^{\frac{1}{3}}\right) . \tag{7.18}$$

To derive the optimal process count, consider $V + C = c$ and $H(P) = \alpha + \frac{1-\alpha}{P}$ for $\alpha > 0$. Then, Equation (7.11) can be expanded as

$$\mathbb{H}_3^{\text{prc}}(P) = \alpha + 3\alpha \left(\frac{3}{4}(\lambda c)^2 P\right)^{\frac{1}{3}} + \frac{1-\alpha}{P} + o(\lambda^{\frac{2}{3}}) \,. \tag{7.19}$$

The optimal overhead is achieved by setting

$$\frac{\partial \mathbb{H}_3^{\text{prc}}(P)}{\partial P} = \alpha \left(\frac{3}{4}(\lambda c)^2 \frac{1}{P^2}\right)^{\frac{1}{3}} - \frac{1-\alpha}{P^2} = 0 \,,$$

which gives rise to $P^* = \left(\frac{4}{3}\left(\frac{1-\alpha}{\alpha}\right)^3 \left(\frac{1}{c\lambda}\right)^2\right)^{\frac{1}{4}}$. Now, the optimal process count is upper-bounded by $\frac{Q}{3}$. Thus, P_{opt} is given by Equation (7.14), which again holds true when $c = 0$ or $\alpha = 0$, and the optimal expected speedup satisfies $\mathbb{S}_3^{\text{prc}}(P_{\text{opt}}) = \frac{1}{\mathbb{H}_3^{\text{prc}}(P_{\text{opt}})}$, as shown in Equation (7.16). □

Remarks. For fully parallelizable applications, i.e., $\alpha = 0$, the optimal pattern on a Q-process platform is characterized by

$$P_{\text{opt}} = \frac{Q}{3} \,, \quad T_{\text{opt}} = \begin{cases} \sqrt[3]{\frac{c}{2\lambda^2 Q}} & \text{for } V + C = c \\ \sqrt[3]{\frac{3d}{2\lambda^2 Q^2}} & \text{for } V + C = \frac{d}{P} \end{cases} \,,$$

$$\mathbb{S}_2^{\text{prc}}(P_{\text{opt}}) = \begin{cases} \frac{Q}{3\left(1+3\sqrt[3]{\left(\frac{\lambda c}{2}\right)^2 Q}\right)} & \text{for } V + C = c \\ \frac{Q}{3\left(1+3\sqrt[3]{\left(\frac{3\lambda c}{2}\right)^2 \frac{1}{Q}}\right)} & \text{for } V + C = \frac{d}{P} \end{cases} \,.$$

Compared with duplication, the ability to correct errors in triplication allows checkpoints to be taken less frequently (i.e., larger checkpointing period). In terms of the expected speedup, triplication suffers from a smaller error-free speedup ($\frac{Q}{3}$ vs $\frac{Q}{2}$) due to the use of fewer concurrent processes to perform useful work, but also has a smaller error-induced denominator, especially on platforms with a large number of processes Q. In Section 7.7, we will conduct simulations to evaluate this trade-off and compare the performance of duplication and triplication.

7.5.3 General process replication

In this section, we consider a general resilience framework and derive the optimal pattern using n replicas per process, where n is an arbitrary constant. Moreover, let k denote the number of "good" replicas (not hit by silent errors) that is required to reach a consensus through voting. Optimistically, assuming any two replicas that are hit by silent errors will produce different results, we can set $k = 2$, i.e., at least two replicas should agree on the result to avoid a rollback. Under a more pessimistic assumption, we will need a majority of the n replicas to agree on the result, so in this case we need $k = \lfloor \frac{n}{2} \rfloor + 1$. Our results are independent of the choice of k.

As for duplication and triplication, for a given (n, k) pair, we can compute the failure probability of a pattern with length T and P processes as follows:

$$\begin{aligned}\mathbb{P}^{\text{prc}}_{n,k}(T,P) &= 1 - \mathbb{P}(\text{“No process fails”}) \\ &= 1 - \left(1 - \mathbb{P}^{\text{prc}}_{n,k}(T,1)\right)^{P}, \end{aligned} \tag{7.20}$$

where

$$\begin{aligned}\mathbb{P}^{\text{prc}}_{n,k}(T,1) &= \sum_{j=0}^{k-1} \binom{n}{j} \left(1 - \mathbb{P}^{\text{prc}}_{1}(T,1)\right)^{j} \mathbb{P}^{\text{prc}}_{1}(T,1)^{n-j} \\ &= \sum_{j=0}^{k-1} \binom{n}{j} e^{-\lambda j T} \left(1 - e^{-\lambda T}\right)^{n-j} \end{aligned} \tag{7.21}$$

denotes the failure probability of a single process with n replicas due to less than k of them surviving silent errors.

The following theorem shows the general result for (n, k)-process replication.

Theorem 27. *On a system with a total number of Q available processors, a first-order approximation to the optimal number of processes for an application with n replicas per process (k of which must concur to avoid a rollback) is given by*

$$P_{\text{opt}} = \min \left\{ \frac{Q}{n}, \left(\gamma_{n,k} \left(\frac{1-\alpha}{\alpha} \right)^{n-k+2} \left(\frac{1}{c\lambda} \right)^{n-k+1} \right)^{\frac{1}{n-k+3}} \right\} . \tag{7.22}$$

The associated optimal checkpointing period and the expected speedup function of the application are

$$T_{\text{opt}}(P_{\text{opt}}) = \left(\frac{V+C}{\beta_{n,k} \lambda^{n-k+1} P_{\text{opt}}} \right)^{\frac{1}{n-k+2}} , \tag{7.23}$$

$$\mathbb{S}^{\text{prc}}_{n,k}(P_{\text{opt}}) = \frac{S(P_{\text{opt}})}{1 + (n-k+2) \left(\frac{((V+C)\lambda)^{n-k+1} P_{\text{opt}}}{\gamma_{n,k}} \right)^{\frac{1}{n-k+2}}} . \tag{7.24}$$

Here, $\beta_{n,k} = \binom{n}{k-1}(n-k+1)$ and $\gamma_{n,k} = \frac{(n-k+1)^{n-k+1}}{\binom{n}{k-1}}$.

Proof. As in the preceding two cases, we start by approximating the error probability. First, we can approximate the probability of single process failure as

$$\begin{aligned}\mathbb{P}^{\text{prc}}_{n,k}(T,1) &= \sum_{j=0}^{k-1} \binom{n}{j} (1-\lambda T)^{j} (\lambda T)^{n-j} \\ &\approx \binom{n}{k-1} (\lambda T)^{n-k+1} + o((\lambda T)^{n-k+1}) . \end{aligned}$$

We can now approximate

$$\begin{aligned}\frac{\mathbb{P}^{\text{prc}}_{n,k}(T,P)}{1-\mathbb{P}^{\text{prc}}_{n,k}(T,P)} &\approx \left(\frac{1}{1-\mathbb{P}^{\text{prc}}_{n,k}(T,1)}\right)^P - 1 \\ &\approx \left(1+\binom{n}{k-1}(\lambda T)^{n-k+1}\right)^P - 1 \\ &= \sum_{j=0}^{P}\binom{P}{j}\left(\binom{n}{k-1}(\lambda T)^{n-k+1}\right)^j - 1 \\ &= \binom{n}{k-1}P(\lambda T)^{n-k+1} + o(P(\lambda T)^{n-k+1}) \,.\end{aligned}$$

Thus, the expected execution time of a pattern can be expressed as

$$\begin{aligned}\mathbb{E}^{\text{grp}}_n(T,P)k = T+V+C+\binom{n}{k-1}P(\lambda T)^{n-k+1}(T+V+R) \\ + o(\lambda^{n-k+1}PT^{n-k+2}) \,.\end{aligned}$$

The derivation of the optimal pattern then follows exactly the same steps as in the proofs of Theorems 25 and 26, and the detailed derivation steps are omitted here. □

Remarks. Theorem 27 encompasses Theorem 25 and Theorem 26 as special cases. We point out that it even holds for the case without replication, i.e., when $n = k = 1$. In this case, Theorem 27 evaluates to

$$\begin{aligned}T_{\text{opt}}(P) &= \sqrt{\frac{V+C}{\lambda P}} \,, \\ \mathbb{S}^{\text{prc}}_1(P) &= \frac{S(P)}{1+2\sqrt{(V+C)\lambda P}} \,,\end{aligned}$$

which is consistent with the results obtained in Chapter 4, provided that a reliable silent error detector is available. However, as mentioned previously, such a detector is only known in some application-specific domains. For general-purpose computations, replication appears to be the only viable approach to detect/correct silent errors so far.

7.6 Group replication

In this section, we consider group replication. Recall that, unlike process replication where the results of each process from different replicas are independently compared, group replication compares the outputs of the different groups viewed as independent black-box applications. First, we make the following technical observation, which establishes the relationship between the two replication mechanisms from the resilience point of view.

Observation 4. *Running an application using group replication with n replicas, where each replica has P processes and each process has error rate λ, has the same failure probability as running it using process replication with one process, which has error rate λP and is replicated n times.*

The above observation allows us to compute the failure probability for group replication by deriving from the corresponding formulas under process replication while setting $P = 1$ and $\lambda = \lambda P$. The rest of this section shows the results for duplication, triplication, and a general group replication framework. Proofs are similar to those in process replication, and are hence omitted.

7.6.1 Group duplication

By applying Observation 4 on Lemma 12, we can get the failure probability for a given pattern under group duplication as follows.

Lemma 14. *Using group duplication, the failure probability of a computing pattern of length T and with P processes is given by*

$$\mathbb{P}_2^{\text{grp}}(T, P) = 1 - e^{-2\lambda TP} . \tag{7.25}$$

This leads us to the following theorem on the optimal pattern:

Theorem 28. *A first-order approximation to the optimal number of processes for an application with 2 replica groups is given by*

$$P_{\text{opt}} = \min \left\{ \frac{Q}{2}, \left(\frac{1}{2} \left(\frac{1-\alpha}{\alpha} \right)^2 \frac{1}{c\lambda} \right)^{\frac{1}{3}} \right\} , \tag{7.26}$$

where Q denotes the total number of available processes in the system. The associated optimal checkpointing period and the expected speedup function of the application are

$$T_{\text{opt}}(P_{\text{opt}}) = \left(\frac{V+C}{2\lambda P_{\text{opt}}} \right)^{\frac{1}{2}} , \tag{7.27}$$

$$\mathbb{S}_2^{\text{grp}}(P_{\text{opt}}) = \frac{S(P_{\text{opt}})}{1 + 2\big(2\lambda(V+C)P_{\text{opt}}\big)^{\frac{1}{2}}} . \tag{7.28}$$

Remarks. The result is identical to that of process duplication. Indeed, in both cases, a single silent error that strikes any of the running processes will cause the whole application to fail.

7.6.2 Group triplication

Again, applying Observation 4 on Lemma 13, we can get the failure probability for a given pattern under group triplication.

Lemma 15. *Using group triplication, the failure probability of a computing pattern of length T and with P processes is given by*

$$\mathbb{P}_3^{\text{grp}}(T,P) = 1 - \left(3e^{-2\lambda TP} - 2e^{-3\lambda TP}\right) . \tag{7.29}$$

The following theorem shows the optimal pattern.

Theorem 29. *A first-order approximation to the optimal number of processes for an application with 3 replica groups is given by*

$$P_{\text{opt}} = \min\left\{\frac{Q}{3}, \left(\frac{1}{6}\left(\frac{1-\alpha}{\alpha}\right)^3\left(\frac{1}{c\lambda}\right)^2\right)^{\frac{1}{5}}\right\} , \tag{7.30}$$

where Q denotes the total number of available processes in the system. The associated optimal checkpointing period and the expected execution overhead are

$$T_{\text{opt}}(P_{\text{opt}}) = \left(\frac{V+C}{6(\lambda P_{\text{opt}})^2}\right)^{\frac{1}{3}} , \tag{7.31}$$

$$\mathbb{S}_3^{\text{grp}}(P_{\text{opt}}) = \frac{S(P_{\text{opt}})}{1+3\left(\frac{3}{4}\left(\lambda(V+C)P_{\text{opt}}\right)^2\right)^{\frac{1}{3}}} . \tag{7.32}$$

Remarks. Compared to the result of process triplication (Theorem 26) and under the same condition (e.g., $\alpha = 0$ so both scenarios allocate the same number of $P_{\text{opt}} = \frac{Q}{3}$ processes to each replica), group triplication needs to place checkpoints more frequently yet enjoys a smaller execution speedup. This provides a theoretical explanation to the common understanding that group replication in general cannot recover from some error combinations that its process counterpart is capable of, making the latter a superior replication mechanism provided that it can be feasibly implemented.

7.6.3 General group replication

Finally, we consider a general group replication framework and derive the optimal pattern using a constant number of n replica groups, out of which k of them must agree to avoid a rollback. Again, the results work for any choice of k.

Now, applying Observation 4 on Equations (7.20) and (7.21), we can compute the failure probability of a pattern with length T and P processes under a (n,k) group replication model:

$$\mathbb{P}_{n,k}^{\text{grp}}(T,P) = \sum_{j=0}^{k-1}\binom{n}{j}\left(e^{-\lambda PT}\right)^j\left(1-e^{-\lambda PT}\right)^{n-j} . \tag{7.33}$$

The following theorem shows the general result for this case.

Theorem 30. *On a system with a total number of Q available processors, a first-order approximation to the optimal number of processes for an application with n replica groups (k of which must concur to avoid a rollback) is given by*

$$P_{\text{opt}} = \min \left\{ \frac{Q}{n}, \left(\frac{1}{\beta_{n,k}} \left(\frac{1-\alpha}{\alpha} \right)^{n-k+2} \left(\frac{1}{c\lambda} \right)^{n-k+1} \right)^{\frac{1}{2n-2k+3}} \right\} . \tag{7.34}$$

The associated optimal checkpointing period and the expected speedup function of the application are

$$T_{\text{opt}}(P_{\text{opt}}) = \left(\frac{C+V}{\beta_{n,k}(\lambda P_{\text{opt}})^{n-k+1}} \right)^{\frac{1}{n-k+2}} , \tag{7.35}$$

$$\mathbb{S}^{\text{grp}}_{n,k}(P_{\text{opt}}) = \frac{S(P_{\text{opt}})}{1 + (n-k+2)\left(\frac{1}{\gamma_{n,k}} \left((V+C)\lambda P_{\text{opt}} \right)^{n-k+1} \right)^{\frac{1}{n-k+2}}} . \tag{7.36}$$

Here, $\beta_{n,k} = \binom{n}{k-1}(n-k+1)$ *and* $\gamma_{n,k} = \frac{(n-k+1)^{n-k+1}}{\binom{n}{k-1}}$.

7.7 Simulations

We conduct a set of simulations whose goal is twofold: (i) validate the accuracy of the theoretical study; and (ii) evaluate the efficiency of both process and group replication under different scenarios at extreme scale. The simulator is publicly available at `http://perso.ens-lyon.fr/aurelien.cavelan/replication.zip` so that interested readers can instantiate their preferred scenarios and repeat the same simulations for reproducibility purpose.

7.7.1 Simulation setup

The simulator has been designed to simulate each process individually, and each process has its own error trace. A simulation works as follows: we feed the simulator with the model parameters μ_{ind}, Q, C, V, R, and α, and we compute the associated optimal number of processes P_{opt} and the optimal checkpointing period $T_{\text{opt}}(P_{\text{opt}})$ using the corresponding model equations. For each run, the simulator outputs the efficiency, defined as $\frac{\mathbb{S}(P_{\text{opt}})}{Q}$, as well as the average number of errors and the average number of recoveries per million CPU hours of work. Then, for each of the following scenarios, we compare the simulated efficiency to the theoretical value, obtained using the model equations for $\mathbb{S}(P_{\text{opt}})$. As suggested by Observation 4, process and group replications with $n = 2$ lead to identical results, so we have merged them together.

In the following, we set the cost of recovery to be the same as the checkpoint cost (as discussed in Section 7.3), and we set the cost $V + C$ according the values of c and d as in Equation (7.2). We consider different Mean Time Between Errors (MTBE), ranging from 10^6 seconds ($\approx$ 11 days) down to 10^2 seconds ($<$ 2 minutes) for $Q = 10^6$ processes, matching the numbers in [89].

7.7.2 Impacts of MTBE and checkpoint cost

Figure 7.1 presents the impact of the $MTBE$ on the efficiency of both duplication and triplication for three different checkpoint costs, but using the same value $\alpha = 10^{-6}$ for the sequential fraction of the application (see next section for the impact of varying α). The first row of plots is obtained with a cost of 30 minutes (i.e. $c = 1,800, d = 0$), the second row with a cost of 60 seconds (i.e. $c = 60, d = 0$), and the last row with $c = 0, d = 10^7$, which correspond to a checkpoint cost of 20 seconds for duplication with $\frac{Q}{2}$ processes and 30 seconds for triplication with $\frac{Q}{3}$ processes. In addition to the efficiency, we provide the average number of errors and recoveries per million hours of work, the optimal checkpointing period $T_{\text{opt}}(P_{\text{opt}})$ and the optimal number of processes P_{opt}.

Efficiency. First, we observe in the first column that the difference between the theoretical efficiency and the simulated efficiency remains small ($< 5\%$ absolute difference), which shows the accuracy of the first-order approximation. Then, with very few errors ($MTBE = 10^6$), we observe that duplication is always better than triplication. This is as expected, since the maximum efficiency for duplication is 0.5 (assuming $\alpha = 0$ and no error), while the maximum efficiency for triplication is 0.33. However, as the $MTBE$ decreases, triplication becomes more attractive and eventually outperforms duplication. With a checkpoint cost of 30 minutes (first row), the $MTBE$ required is around 28 hours for process triplication to win and 20 hours for group triplication to win. With smaller checkpoint costs, such as 60 seconds (second row) and 30 seconds (third row), checkpoints can be more frequent and the $MTBE$ required for triplication to win is pushed down to a couple of hours and a couple of minutes, respectively.

Number of errors and recoveries. The second column presents the number of errors and the corresponding number of recoveries per million hours of work. The number of errors is always higher than the number of recoveries, because multiple errors can occur during a period (before the checkpoint, which is the point of detection), causing a single recovery. At $MTBE = 10^2$, almost half of the errors that occurred with duplication were actually hidden behind another error. Even more errors were hidden with group triplication, since one more error (in a different replica) is required to cause a recovery. Finally, (almost) all errors were hidden with process replication, which is able to handle many errors, as long as they strike in different processes.

Optimal checkpointing period. The third column shows the optimal length of the pattern. In order to cope with the increasing number of errors and recoveries, the length of the optimal period becomes smaller. Note that the length of the period for group triplication is comparable to that for duplication, around one day when $MTBE = 10^6$ down to a couple of minutes when $MTBE = 10^2$. However, the length of the pattern for process triplication is always higher by several orders of magnitude, from more than 10 days when $MTBE = 10^6$ down to a couple of hours when $MTBE = 10^2$.

Optimal number of processes. With $\alpha = 10^{-6}$, the application has ample parallelism, so the optimal number of processes to use is always $\frac{Q}{2} = 5 \cdot 10^5$ for duplication and $\frac{Q}{3} \approx 3.3 \cdot 10^5$ for triplication, except when $MTBE = 10^2$ and $c = 1,800$, where the optimal number of processes for duplication is $\approx 3 \cdot 10^5$ and the optimal number of processes for group triplication is $\approx 2 \cdot 10^5$.

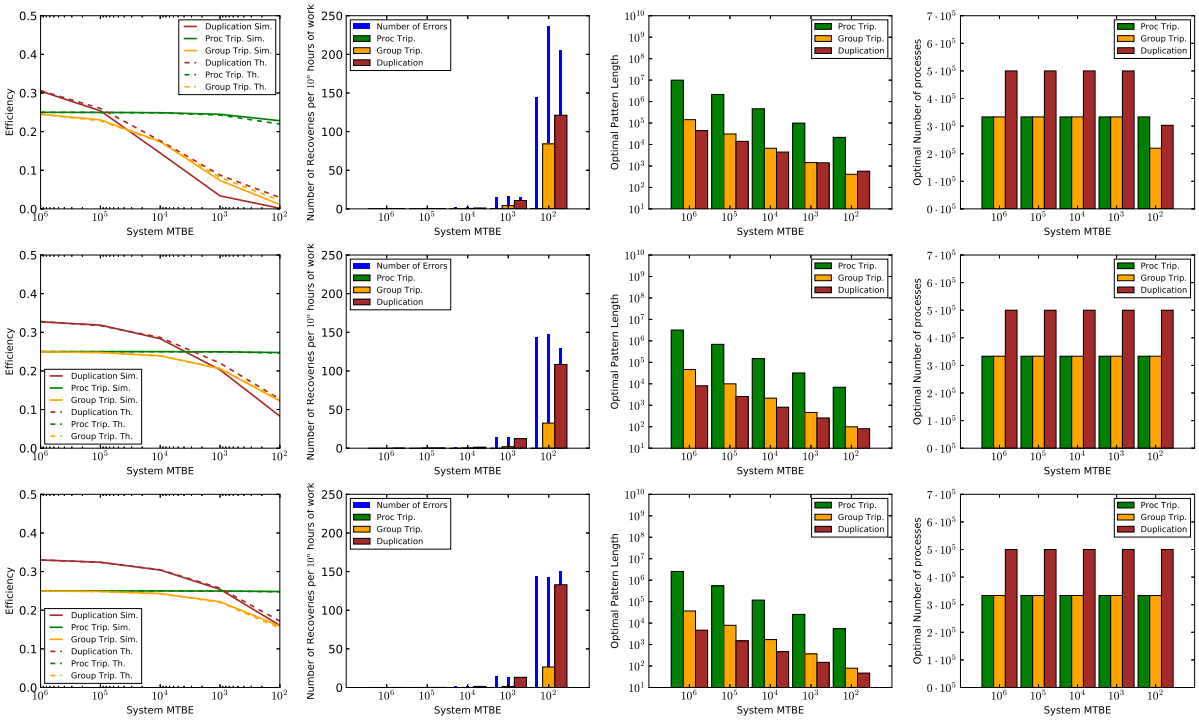


Figure 7.1: Impact of System MTBE on the efficiency with $c=1,800, d=0$ (top), $c=60, d=0$ (middle), $c=0, d=10^7$ (bottom) and $\alpha=10^{-6}$.

7.7.3 Impact of sequential fraction (Amdahl)

Figure 7.2 presents two additional simulation results for $\alpha = 10^{-7}$ and $\alpha = 10^{-5}$. With a small fraction of sequential work (left plots), the efficiency is improved ($\approx 85\%$ of the maximum efficiency for duplication and $\approx 95\%$ for triplication at $MTBE = 10^6$), and both duplication and triplication use all processes available. On the contrary, with a higher sequential fraction of work (right plots), the efficiency drops ($< 20\%$ of the maximum efficiency for duplication and $< 30\%$ for triplication at $MTBE = 10^6$), and using more processes does not improve the efficiency and only contributes to increasing the number of errors. Therefore, these results suggest that even when using replication or triplication, there comes a point where it is no longer beneficial to use all available processes. In this example, when $MTBE = 10^2$, duplication and group triplication would use fewer than $2 \cdot 10^5$ processes (one fifth of the available resources). Process triplication, on the other hand, still utilizes all the resources and outperforms the other two schemes in terms of the efficiency across the whole range of system MTBE.

7.7.4 Impact of number of processes

Figure 7.3 shows the impact of the number of processes on the simulated efficiency of different replication scenarios. In addition, we also show (as big dots) the theoretical efficiency obtained with the optimal number of processes from Theorems 25, 26 and 29. By varying the number of processes, we find that the simulated optimum (that yields the best efficiency) matches our

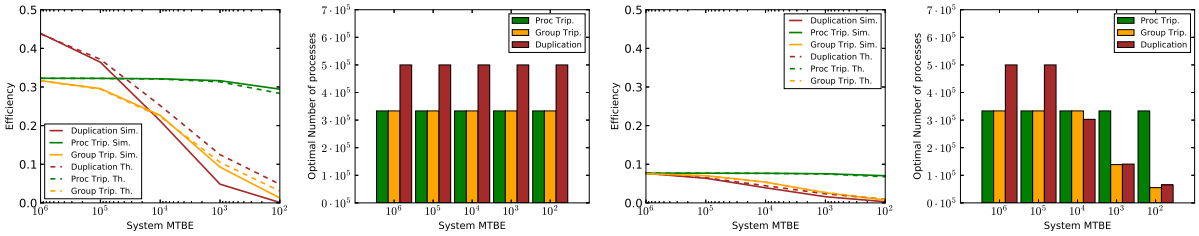


Figure 7.2: Impact of sequential fraction (in Amdahl's Law) on efficiency and optimal number of processes with $\alpha = 10^{-7}$ (left) and $\alpha = 10^{-5}$ (right).

theoretical optimal number of processes closely. We can also see that process triplication scales very well with increasing number of processes. As opposed to group triplication, which has to recover from a checkpoint if just two errors strike in two different replicas, process triplication benefits from the additional process: from a resilience point of view, each process acts as a buffer to handle one more error; in other words, the probability that two errors strike the two replicas of the same process decreases, thereby improving the efficiency.

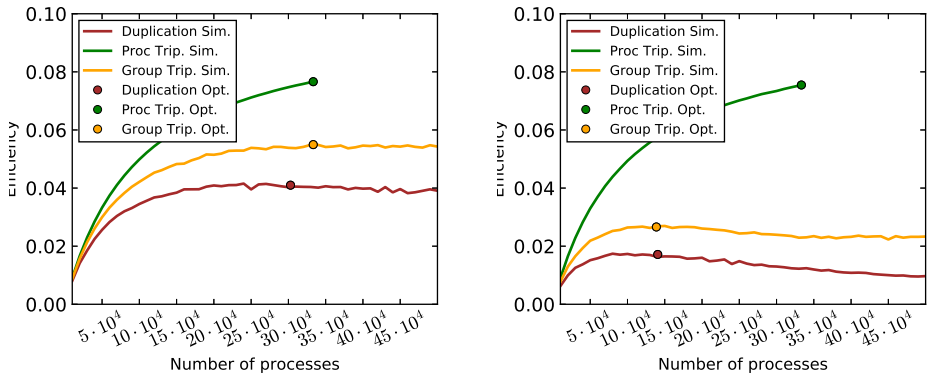


Figure 7.3: Impact of the number of processes on the efficiency with $MTBE = 10^4$ (left), $MTBE = 10^3$ (right), $Q = 10^6$, $c = 1n800, d = 0$, and $\alpha = 10^{-5}$.

7.7.5 Summary

Results suggest that duplication is more efficient than triplication for high $MTBE$ (e.g. 10^5 seconds for $C = 30$ minutes). If process triplication is available, then it is always more efficient for smaller $MTBE$: its efficiency remains stable despite the increasing number of failures. If process triplication is not available, we show that group triplication is slightly more efficient than duplication for small $MTBE$, but the gain is small. Furthermore, the impact of the sequential fraction of work α (in Amdahl's Law) is twofold: it limits the efficiency (e.g. 15% of the maximum with $\alpha = 10^{-5}$ for for both duplication and triplication), and it is a major factor in limiting the optimal number of processes (e.g. one tenth of the platform with $\alpha = 10^{-5}$ and $Q = 10^6$ at $MTBE = 10^2$).

7.8 Conclusion

Silent-errors represent a major threat to the HPC community. In the absence of application-specific detectors, replication is the only solution. Unfortunately, it comes with high cost: by definition, the efficiency is upper-bounded by 0.5 for duplication, and by 0.333 for triplication. Are these upper bounds likely to be achieved? If yes, it means that duplication should always be preferred to triplication. If not, it means that in some scenarios, the striking of errors is so frequent that duplication, and in particular group duplication, is not the right choice.

The major contribution of this chapter is to provide an in-depth analysis of process and group duplication, and of process and group triplication. Given a level n of replication, and a set of application/platform parameters (speedup profile, total number or processes, process MTBE, checkpoint time, etc), we derive closed-form formulas for the optimal period size and optimal resource usage, and for the overall efficiency of the approach. This allows to choose the best value of n. A set of simulations demonstrate the accuracy of the model and analysis. Our computer-algebra sheets and simulator code are made publicly available, so that one can instantiate their preferred scenario. Altogether, this work has laid the foundations for a better understanding of the impact of silent errors on HPC computing at scale.

Conclusion

In this thesis, we have addressed several critical exascale challenges related to resilience. For divisible applications in Part I, we have modeled and analyzed the impact of verifications and partial verifications on the performance. We have proven that the optimal pattern does not contain any imprecise detector (i.e., any detector that generates false-positives), and that using only the detector with the highest *accuracy-to-cost-ratio* is a good approximation in practice. Following these results, we have been able to extend previous work on checkpointing, and most notably the formula obtained by Young and Daly (see Equation (2)) by combining both checkpointing, verification, fail-stop and silent errors into optimal resilience patterns. In addition, we have proposed a new model to derive the optimal checkpointing period for multi-level checkpointing in the presence of fail-stop errors only.

For workflow applications in Part II, our contribution is twofold. On the one hand, we have proposed several sophisticated multi-level dynamic programming algorithms that compute the optimal checkpoint positions for a linear chain of tasks in polynomial time. These results nicely extend the original algorithm proposed by Toueg and Babaoglu for fail-stop errors, as we were able to combine both fail-stop and silent errors with multi-level checkpointing and verifications. On the other hand, we have proposed a new approach to reduce the energy consumption of independent tasks by aggressively lowering the supply voltage below the nominal voltage, thereby introducing timing errors. Based on a formal model of timing errors, we have provided an optimal level algorithm to schedule independent tasks, using verifications to both detect and correct errors, and we have proven its global optimality when voltage switching costs are linear.

Furthermore, we have extended our analysis to compute the optimal number of processors in Part III. We have provided the exact expression for the expected execution time of a pattern, and closed-form first-order approximation formulas to compute the optimal checkpointing period T^* and optimal number of processors P^* when the application speedup profile obeys *Amdahl's law*. In addition, these results are the first that analytically establish the relationship between P^* and T^* as a function of the system MTBF, and they offer new insights into the relationships of Amdhal's law and the Young/Daly approximation formula. Finally, we have studied group replication and process replication as error detection and correction mechanisms. Applying the previous approach to derive the optimal number of processors, we were able to derive closed-form formulas for both the optimal period size and optimal resource usage. This allows to choose the best level of replication (duplication, triplication, or more).

Altogether, we have faced a number of difficulties. When it comes to deriving optimal algorithms and exact formulas, we have often resorted to first or second order approximations with respect to the error rate, which is valid as long as the other resilience parameters are large

in front of this rate. In particular, we have been able to validate our results with an exhaustive set of simulations, showing the accuracy of our models, and comparing our results to the state-of-the-art when it was possible.

Perspectives and future work

In the short term, we plan to generalize the results obtained for replication to address intermediate replication types, such as node replication or thread replication, and by including fail-stop errors into the picture.

Then, a possible direction is to assess the usefulness of replication when applied to application workflows. Indeed, one can choose to replicate some tasks in order to increase the reliability of the execution. However, replication is expensive and it may not be necessary to replicate all the tasks. The problem is then to decide which task to replicate in order to maximize the performance.

In the long term, an interesting future direction is to assess the usefulness of our approach for linear chains of tasks when applied to general application workflows. The problem gets much more challenging, even in the simplified scenario where each task requires the entire platform to execute. In fact, in this simplified scenario, it is already NP-hard to decide which task to checkpoint in a simple join graph ($n-1$ source tasks and a common sink task), with only fail-stop errors striking (hence a single level of checkpoint and no verification at all) [2]. Still, heuristics are urgently needed to address this problem, with several error sources and several checkpoint and verification types, if we are to deploy general HPC workflows efficiently at scale.

Conclusion

Dans cette thèse, nous avons adressé plusieurs aspects critiques de la recherche Exascale sur la résilience. Pour les applications divisibles dans la Partie I, nous avons modélisé et analysé l'impact des vérifications garanties et des vérifications partielles sur la performance. Nous avons prouvé que le schéma de résilience optimal ne contient pas de détecteurs imprécis (c'est à dire qui créer des faux-positifs), et que utiliser seulement le détecteur avec le plus grand *accuracy-to-cost-ratio* est une bonne approximation en pratique. A partir de ces résultats, nous avons été capable d'étendre les travaux précédents sur les checkpoints, et plus particulièrement la formule de Young et Daly (voir l'Equation 2) en combinant checkpoints, vérifications, pannes et erreurs silencieuses dans un même schéma de résilience. Nous avons également proposé un nouveau modèle afin de dériver la période de optimale de checkpoint avec plusieurs niveaux en présence de pannes seulement.

Pour les applications type workflows dans la Partie II, notre contribution est double. D'un côté, nous avons proposé plusieurs algorithmes sophistiqués à base de programmation dynamique afin de calculer la position optimale des checkpoints pour une chaîne de tâche en temps polynomial. Ces résultats étendent l'algorithme original proposé par Toueg et Babaoglu pour les erreurs de type pannes, puisque nous avons été capable de combiner à la fois les pannes et les erreurs silencieuses, les checkpoints et les vérifications. D'autre part, nous avons proposé une nouvelle approche pour réduire la consommation d'énergie pour l'exécution de tâches indépendantes en réduisant le voltage de manière drastique, introduisant dans le même temps des erreurs de synchronisation. Basé sur une définition formelle des erreurs de synchronisation, nous avons proposé un algorithme à niveaux optimal pour ordonnancer des tâches indépendantes, en utilisant des vérifications pour à la fois détecter et corriger les erreurs, et nous avons prouver son optimalité globale quand le coût de changement de voltage est linéaire.

En outre, nous avons ensuite étendu notre analyse pour calculer le nombre optimal de processeurs dans la Partie III. Nous avons proposé une expression exacte pour calculer l'espérance du temps d'exécution d'un schéma de résilience, ainsi que des formes closes, en utilisant une approximation au premier ordre, pour calculer la période de checkpoint optimale T^* et le nombre optimal de processeurs P^* quand le profile d'accélération de l'application obéit à la *loi d'Amdahl*. Ces résultats sont les premiers à établir de manière formelle un lien entre P^* et T^* en fonction du MTBF. Enfin, nous avons étudié la réplication de groupe et la réplication de processus comme mécanismes de détection et de correction. En appliquant les approches précédentes pour dériver le nombre optimal de processeurs. Nous avons été capable de dériver des formes closes pour à la fois la taille de la période et l'utilisation des ressources. Cela nous a permis de déterminer le meilleur niveau de réplication (duplication, triplication, ou plus).

Finalement, nous avons fait face à un certain nombre de difficultés. Lorsque avons du dériver des formules et les algorithmes optimaux, nous avons souvent eu recourt à une approximation au premier ou au deuxième ordre, par rapport au taux d'erreurs, ce qui est valide tant les autres paramètres de résilience sont larges devant le taux d'erreurs. En particulier, nous avons été capable de valider nos résultats avec un jeu de simulations exhaustif, montrant la précision de nos modèles, et en comparant avec l'état de l'art quand cela est possible.

Perspectives et travail future

A court terme, nous prévoyons de généraliser les résultats pour la réplication afin d'adresser des types de réplication intermédiaires, comme la réplication de nuds ou de threads, et en incluant à la fois les pannes et les erreurs silencieuses dans l'analyse.

Ensuite, une direction possible est d'évaluer l'utilité de la réplication quand elle est appliquée à des applications type workflows. En effet, on peut choisir de répliquer quelques tâches afin d'améliorer la fiabilité de l'exécution. Cependant, la réplication est coûteuse et il n'est peut être pas nécessaire de répliquer toutes les tâches. Le problème est alors de décider quelles tâches il faut répliquer pour maximiser les performances.

A long terme, une direction intéressante est l'évaluation de notre approche pour des chaînes de tâches lorsque elle est appliquée à des graphes de tâches. Le problème devient beaucoup plus difficile, même dans un scénario simple où chaque tâche s'exécute sur toute la plateforme. En fait, dans ce scénario simplifié, décider quelle tâche doit être précédée d'un checkpoint dans un simple graphe de type join ($n-1$ sources et un seul puit commun) avec seulement des erreurs de type panne (un seul type de checkpoint et pas de vérification) est déjà un problème NP-difficile. Pourtant, des heuristiques sont nécessaires pour adresser ce problème, avec plusieurs types d'erreurs et de vérifications, si nous voulons déployer des workflows de manière efficace en passant à l'échelle.

Bibliography

[1] G. Amdahl. "The validity of the single processor approach to achieving large scale computing capabilities." In: *AFIPS Conference Proceedings*. Vol. 30. AFIPS Press, 1967, pp. 483–485.

[2] G. Aupy, A. Benoit, H. Casanova, and Y. Robert. "Scheduling computational workflows on failure-prone platforms." In: *17th Workshop on Advances in Parallel and Distributed Computational Models (APDCM'15)*. 2015.

[3] G. Aupy, A. Benoit, T. Hérault, Y. Robert, F. Vivien, and D. Zaidouni. "On the combination of silent error detection and checkpointing." In: *Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*. 2013, pp. 11–20.

[4] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni. "Checkpointing algorithms and fault prediction." In: *J. Parallel and Distributed Computing* 74.2 (2014), pp. 2048–2064.

[5] A. Avizienis, J. Laprie, B. Randell, and C. E. Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing." In: *IEEE Trans. Dependable Sec. Comput.* 1.1 (2004), pp. 11–33.

[6] P. Balaprakash, L. A. B. Gomez, M.-S. Bouguerra, S. M. Wild, F. Cappello, and P. D. Hovland. "Analysis of the Tradeoffs Between Energy and Run Time for Multilevel Checkpointing." In: *Proc. PMBS'14*. 2014.

[7] N. Bansal, T. Kimbrel, and K. Pruhs. "Speed Scaling to Manage Energy and Temperature." In: *Journal of the ACM* 54.1 (2007), pp. 1–39.

[8] L. Bautista Gomez and F. Cappello. "Detecting and Correcting Data Corruption in Stencil Applications through Multivariate Interpolation." In: *Proceedings of the 1st International Workshop on Fault Tolerant Systems*. FTS'15. Chicago, USA: IEEE, 2015.

[9] L. Bautista Gomez and F. Cappello. "Detecting Silent Data Corruption Through Data Dynamic Monitoring for Scientific Applications." In: *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '14. Orlando, Florida, USA: ACM, 2014, pp. 381–382.

[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] L. Bautista Gomez and F. Cappello. "Exploiting Spatial Smoothness in HPC Applications to Detect Silent Data Corruption." In: *Proceedings of the 17th IEEE International Conference on High Performance Computing and Communications*. HPCC'15. New York, USA: IEEE, 2015.

[12] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. "FTI: High Performance Fault Tolerance Interface for Hybrid Systems." In: *Proc. SC'11*. 2011.

[13] A. Benoit, S. K. Raina, and Y. Robert. "Efficient checkpoint/verification patterns." In: *The International Journal of High Performance Computing Applications (IJHPCA)* 31.1 (2017), pp. 52–65.

[14] A. R. Benson, S. Schmit, and R. Schreiber. "Silent error detection in numerical time-stepping schemes." In: *Int. J. High Performance Computing Applications* DOI: 10.1177/109434201453 (2014).

[15] 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: *Proceedings of The ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. HPDC '15. 2015.

[16] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. "Algorithm-based fault tolerance applied to high performance computing." In: *Journal of Parallel and Distributed Computing* 69.4 (2009), pp. 410–416. ISSN: 0743-7315.

[17] G. Bosilca et al. "Unified model for assessing checkpointing protocols at extreme-scale." In: *Concurrency and Computation: Practice and Experience* (2013).

[18] S. Boyd and L. Vandenberghe. *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004. ISBN: 0521833787.

[19] P. G. Bridges, K. B. Ferreira, M. A. Heroux, and M. Hoemmen. "Fault-tolerant iterative methods via selective reliability." In: *ArXiv e-prints* (2012).

[20] A. G. Bromley. "Charles Babbage's Analytical Engine, 1838." In: *IEEE Annals of the History of Computing* 4.3 (1982), pp. 196–217.

[21] G. Bronevetsky and B. de Supinski. "Soft error vulnerability of iterative linear algebra methods." In: *Proceedings of the International Conference on Supercomputing (ICS)*. 2008, pp. 155–164.

[22] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors." In: *IEEE Micro* 20.6 (2000), pp. 26–44.

[23] F. Cappello, E. M. Constantinescu, P. D. Hovland, T. Peterka, C. Phillips, M. Snir, and S. M. Wil. *Improving the trust in results of numerical simulations and scientific data analytics*. White paper MCS-TM-352. ANL, 2015.

[24] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. "Toward Exascale Resilience." In: *Int. Journal of High Performance Computing Applications* 23.4 (2009), pp. 374–388.

[25] 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).

[26] H. Casanova, M. Bougeret, Y. Robert, F. Vivien, and D. Zaidouni. “Using group replication for resilience on exascale systems.” In: *Int. Journal of High Performance Computing Applications* 28.2 (2014), pp. 210–224.

[27] H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni. “On the impact of process replication on executions of large-scale parallel applications with coordinated checkpointing.” In: *Future Generation Comp. Syst.* 51 (2015), pp. 7–19.

[28] K. M. Chandy and L. Lamport. “Distributed Snapshots: Determining Global States of Distributed Systems.” In: *ACM Transactions on Computer Systems* 3.1 (1985), pp. 63–75.

[29] J.-J. Chen and C.-F. Kuo. “Energy-Efficient Scheduling for Real-Time Systems on Dynamic Voltage Scaling (DVS) Platforms.” In: *Proc. Int. Works. on Real-Time Computing Systems and Applications*. 2007.

[30] Z. Chen. “Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods.” In: *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2013, pp. 167–176.

[31] E. Ciocca, I. Koren, Z. Koren, C. M. Krishna, and D. S. Katz. “Application-Level Fault Tolerance in the Orbital Thermal Imaging Spectrometer.” In: *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC’04)*. Papeete, Tahiti, French Polynesia, 2004, pp. 43–48.

[32] E. Ciocca, I. Koren, and C. M. Krishna. “Determining acceptance tests for application-level fault detection.” In: *Proceedings of the 2nd ASPLOS EASY Workshop*. 2002, pp. 47–53.

[33] R. Cohen and L. Katzir. “The Generalized Maximum Coverage Problem.” In: *Inf. Process. Lett.* 108.1 (2008), pp. 15–22.

[34] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[35] S. P. Crago, D. I. Kang, M. Kang, R. Kost, K. Singh, J. Suh, and J. P. Walters. “Programming Models and Development Software for a Space-Based Many-Core Processor.” In: *4th Int. Conf. onon Space Mission Challenges for Information Technology*. IEEE, 2011, pp. 95–102.

[36] J. T. Daly. “A higher order estimate of the optimum checkpoint interval for restart dumps.” In: *Future Generation Comp. Syst.* 22.3 (2006), pp. 303–312.

[37] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello. “Optimization of multi-level checkpoint model for large scale HPC applications.” In: *Proc. IPDPS’14*. 2014.

[38] S. Di and F. Cappello. “Adaptive Impact-Driven Detection of Silent Data Corruption for HPC Applications.” In: *IEEE Trans. Parallel Distributed Systems* (2016).

[39] S. Di, Y. Robert, F. Vivien, and F. Cappello. “Toward an Optimal Online Checkpoint Solution under a Two-Level HPC Checkpoint Model.” In: *IEEE Trans. Parallel & Distributed Systems* (2016).

[40] J. Dongarra and et al. “The International Exascale Software Project Roadmap.” In: *Int. J. High Perform. Comput. Appl.* 25.1 (2011), pp. 3–60.

[41] J. Dongarra et al. “The International Exascale Software Project: a Call To Cooperative Action By the Global High-Performance Community.” In: *Int. J. High Performance Computing Applications* 23.4 (2009), pp. 309–322.

[42] J. Dongarra, T. Hérault, and Y. Robert. “Performance and reliability trade-offs for the double checkpointing algorithm.” In: *Int. J. of Networking and Computing* 4.1 (2014), pp. 23–41.

[43] M. Dow. “Explicit inverses of Toeplitz and associated matrices.” In: *ANZIAM J.* 44.E (2003), E185–E215.

[44] J. Elliott, M. Hoemmen, and F. Mueller. “Evaluating the Impact of SDC on the GMRES Iterative Solver.” In: *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IPDPS ’14. 2014, pp. 1193–1202.

[45] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. “Combining partial redundancy and checkpointing for HPC.” In: *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2012, pp. 615–626.

[46] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. “A survey of rollback-recovery protocols in message-passing systems.” In: *ACM Computing Survey* 34 (3 2002), pp. 375–408.

[47] E. N. Elnozahy and J. Plank. “Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery.” In: *IEEE Transactions on Dependable and Secure Computing* 1.2 (2004), pp. 97–108.

[48] E. N. Elnozahy and J. Plank. “Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery.” In: *IEEE Trans. Dependable and Secure Computing* 1.2 (2004), pp. 97–108.

[49] C. Engelmann, H. H. Ong, and S. L. Scorr. “The case for modular redundancy in large-scale highh performance computing systems.” In: *PDCN*. IASTED, 2009.

[50] C. Engelmann and B. Swen. “Redundant execution of HPC applications with MR-MPI.” In: *PDCN*. IASTED, 2011.

[51] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner. “Razor: circuit-level correction of timing errors for low-power operation.” In: *IEEE Micro* 24.6 (2004), pp. 10–20.

[52] K. Ferreira, J. Stearley, J. H. I. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. “Evaluating the Viability of Process Replication Reliability for Exascale Systems.” In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. Seattle, WA, 2011, 44:1–44:12.

[53] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. “Detection and correction of silent data corruption for large-scale high-performance computing.” In: *Proc. SC’12*. 2012, p. 78.

[54] F. Firouzi, M. E. Salehi, F. Wang, and S. M. Fakhraie. “An accurate model for soft error rate estimation considering dynamic voltage and frequency scaling effects.” In: *Microelectronics Reliability* 51.2 (2011), pp. 460–467.

[55] R. G. Gallager. *Stochastic Processes: Theory for Applications*. New York, NY, USA: Cambridge University Press, 2014.

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

[57] R. Ge, X. Feng, and K. W. Cameron. “Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters.” In: *Proceedings of SC’05*. 2005, p. 34.

[58] A. Geist. “How to kill a supercomputer: Dirty power, cosmic rays, and bad solder.” In: *IEEE Spectrum* (2016).

[59] C. George and S. S. Vadhiyar. “ADFT: An Adaptive Framework for Fault Tolerance on Large Scale Systems using Application Malleability.” In: *Procedia Computer Science* 9 (2012), pp. 166–175.

[60] D. Hakkarinen and Z. Chen. “Multilevel Diskless Checkpointing.” In: *IEEE Transactions on Computers* 62.4 (2013), pp. 772–783.

[61] E. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, and F. Cappello. “Modeling and tolerating heterogeneous failures in large parallel systems.” In: *Proc. ACM/IEEE Supercomputing’11*. ACM Press, 2011.

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

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

[64] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. “Cosmic rays don’t strike twice: understanding the nature of DRAM errors and the implications for system design.” In: *SIGARCH Comput. Archit. News* 40.1 (2012), pp. 111–122.

[65] H. Jin, Y. Chen, H. Zhu, and X.-H. Sun. “Optimizing HPC Fault-Tolerant Environment: An Analytical Approach.” In: *Proc. ICPP’10*. 2010.

[66] G. Karakonstantis and K. Roy. “Voltage over-scaling: A cross-layer design perspective for energy efficient systems.” In: *European Conference on Circuit Theory and Design (ECCTD)*. 2011, pp. 548–551.

[67] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[68] P. Krause and I. Polian. “Adaptive voltage over-scaling for resilient applications.” In: *Design, Automation Test in Europe Conference Exhibition (DATE)*. 2011, pp. 1–6.

[69] T. Leblanc, R. Anand, E. Gabriel, and J. Subhlok. “VolpexMPI: An MPI Library for Execution of Parallel Applications on Volatile Nodes.” In: *16th European PVM/MPI Users’ Group Meeting*. Springer-Verlag, 2009, pp. 124–133.

[70] G. Lu, Z. Zheng, and A. A. Chien. “When is Multi-version Checkpointing Needed?” In: *Proc. 3rd Workshop on Fault-tolerance for HPC at extreme scale (FTXS)*. 2013, pp. 49–56.

[71] 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.

[72] R. E. Lyons and W. Vanderkulk. “The use of triple-modular redundancy to improve computer reliability.” In: *IBM J. Res. Dev.* 6.2 (1962), pp. 200–209.

[73] E. Meneses, X. Ni, T. Jones, and D. Maxwell. “Analyzing the Interplay of Failures and Workload on a Leadership-Class Supercomputer.” In: *computing* 2.3 (2015), p. 4.

[74] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. “Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System.” In: *Proc. of the ACM/IEEE SC Conf.* 2010, pp. 1–11.

[75] X. Ni, E. Meneses, N. Jain, and L. V. Kalé. “ACR: Automatic Checkpoint/Restart for Soft and Hard Error Protection.” In: *Proc. SC’13*. ACM, 2013.

[76] T. O’Gorman. “The effect of cosmic rays on the soft error rate of a DRAM at ground level.” In: *IEEE Trans. Electron Devices* 41.4 (1994), pp. 553–557.

[77] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth. “Modeling the Impact of Checkpoints on Next-Generation Systems.” In: *24th IEEE Conf. Mass Storage Systems and Technologies*. IEEE, 2007.

[78] J. Plank, K. Li, and M. Puening. “Diskless checkpointing.” In: *IEEE Trans. Parallel Dist. Systems* 9.10 (1998), pp. 972–986. ISSN: 1045-9219.

[79] F. Quaglia. “A Cost Model for Selecting Checkpoint Positions in Time Warp Parallel Simulation.” In: *IEEE Trans. Parallel Dist. Syst.* 12.4 (2001), pp. 346–362.

[80] S. Ramasubramanian, S. Venkataramani, A. Parandhaman, and A. Raghunathan. “Relax-and-Retime: A methodology for energy-efficient recovery based design.” In: *Design Automation Conference (DAC)*. 2013, pp. 1–6.

[81] A. Randall. “The Eckert tapes: Computer pioneer says ENIAC team couldnt afford to fail–and didnt.” In: *Computerworld* 40.8 (2006), p. 18.

[82] M. W. Rashid and M. C. Huang. “Supporting highly-decoupled thread-level redundancy for parallel programs.” In: *14th Int. Conf. on High-Performance Computer Architecture (HPCA)*. IEEE, 2008, pp. 393–404.

[83] 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*. Ed. by A. Y. Zomaya and Y. C. Lee. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2012.

[84] R. Rojas and U. Hashagen. *The First Computers: History and Architectures*. History of computing. MIT Press, 2002. ISBN: 9780262681377.

[85] P. Sao and R. Vuduc. “Self-stabilizing Iterative Solvers.” In: *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. 2013.

[86] B. Schroeder and G. A. Gibson. “Understanding Failures in Petascale Computers.” In: *Journal of Physics: Conference Series* 78.1 (2007).

[87] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. “Fault Tolerant Preconditioned Conjugate Gradient for Sparse Linear System Solution.” In: *Proceedings of the ACM International Conference on Supercomputing (ICS)*. 2012, pp. 69–78.

[88] L. Silva and J. Silva. “Using two-level stable storage for efficient checkpointing.” In: *IEE Proceedings - Software* 145.6 (1998), pp. 198–202.

[89] M. Snir and et al. “Addressing Failures in Exascale Computing.” In: *Int. J. High Perform. Comput. Appl.* 28.2 (2014), pp. 129–173.

[90] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi. “Memory errors in modern systems: The good, the bad, and the ugly.” In: *ACM SIGPLAN Notices*. Vol. 50. 4. ACM. 2015, pp. 297–310.

[91] J. Stearley, K. B. Ferreira, D. J. Robinson, J. Laros, K. T. Pedretti, D. Arnold, P. G. Bridges, and R. Riesen. “Does partial replication pay off?” In: *FTXS*. IEEE, 2012.

[92] *Top500 Supercomputer Sites*. http://www.top500.org.

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

[94] N. H. Vaidya. “A Case for Two-level Distributed Recovery Schemes.” In: *SIGMETRICS Perform. Eval. Rev.* 23.1 (1995), pp. 64–73.

[95] L. Wang, G. von Laszewski, J. Dayal, and F. Wang. “Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS.” In: *Proceedings of IEEE/ACM CCGRID*. 2010.

[96] S. Yi, D. Kondo, B. Kim, G. Park, and Y. Cho. “Using Replication and Checkpointing for Reliable Task Management in Computational Grids.” In: *SC*. ACM, 2010.

[97] J. W. Young. “A first order approximation to the optimum checkpoint interval.” In: *Comm. of the ACM* 17.9 (1974), pp. 530–531.

[98] J. Yu, D. Jian, Z. Wu, and H. Liu. “Thread-level redundancy fault tolerant CMP based on relaxed input replication.” In: *ICCIT*. IEEE, 2011.

[99] G. Zheng, L. Shi, and L. V. Kale. “FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI.” In: *Cluster Computing (CLUSTER)*. IEEE Computer Society, 2004, pp. 93–103.

[100] Z. Zheng and Z. Lan. “Reliability-aware scalability models for high performance computing.” In: *Cluster Computing*. IEEE, 2009.

[101] Z. Zheng, L. Yu, and Z. Lan. “Reliability-Aware Speedup Models for Parallel Applications with Coordinated Checkpointing/Restart.” In: *IEEE Trans. Computers* 64.5 (2015), pp. 1402–1415.

[102] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, and B. Chin. “IBM Experiments in Soft Fails in Computer Electronics.” In: *IBM J. Res. Dev.* 40.1 (1996), pp. 3–18.

[103] J. Ziegler, M. Nelson, J. Shell, R. Peterson, C. Gelderloos, H. Muhlfeld, and C. Montrose. “Cosmic ray soft error rates of 16-Mb DRAM memory chips.” In: *IEEE Journal of Solid-State Circuits* 33.2 (1998), pp. 246–252.

Publications[6]

Book Chapters

[P1] G. Aupy, A. Benoit, A. Cavelan, M. Fasi, Y. Robert, H. Sun, and B. Uçar. "Coping with silent errors in HPC applications." In: *Emergent Computation*. Ed. by A. Adamatzky. Bristol, UK: Springer, 2016, pp. 269–292.

Articles in International Refereed Journals

[J1] L. Bautista-Gomez, A. Benoit, A. Cavelan, S. K. Raina, Y. Robert, and H. Sun. "Coping with recall and precision of soft error detectors." In: *Journal of Parallel and Distributed Computing* 98 (2016), pp. 8–24.

[J2] A. Benoit, A. Cavelan, V. Le Fèvre, Y. Robert, and H. Sun. "Towards Optimal Multi-Level Checkpointing." In: *IEEE Transactions on Computers* (2016).

[J3] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. "Assessing general-purpose algorithms to cope with fail-stop and silent errors." In: *ACM Transactions on Parallel Computing* 3.2 (2016), p. 13.

[J4] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. "Multi-level checkpointing and silent error detection for linear workflows." In: *Journal of Computational Science* (2017).

Articles in International Refereed Conferences

[C1] L. Bautista-Gomez, A. Benoit, A. Cavelan, S. K. Raina, Y. Robert, and H. Sun. "Which verification for soft error detection?" In: *International Conference on High Performance Computing (HiPC)*. IEEE. 2015, pp. 2–11.

[C2] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. "Optimal resilience patterns to cope with fail-stop and silent errors." In: *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2016, pp. 202–211.

[C3] A. Cavelan, J. Li, Y. Robert, and H. Sun. "When Amdahl Meets Young/Daly." In: *Cluster Computing (CLUSTER)*. IEEE. 2016, pp. 203–212.

[C4] A. Cavelan, S. K. Raina, Y. Robert, and H. Sun. "Assessing the impact of partial verifications against silent data corruptions." In: *International Conference on Parallel Processing (ICPP)*. IEEE. 2015, pp. 440–449.

[6] Authors are listed in alphabetical order.

[C5] A. Cavelan, Y. Robert, H. Sun, and F. Vivien. “Scheduling Independent Tasks with Voltage Overscaling.” In: *Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE. 2015, pp. 32–41.

Articles in International Refereed Workshops

[W1] A. Benoit, A. Cavelan, F. Cappello, P. Raghavan, Y. Robert, and H. Sun. “Identifying the right replication level to detect and correct silent errors at scale.” In: *Proceedings of the 7th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. 2017.

[W2] A. Benoit, A. Cavelan, V. Le Fèvre, and Y. Robert. “Optimal checkpointing period with replicated execution on heterogeneous platforms.” In: *Proceedings of the 7th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. 2017.

[W3] A. Benoit, A. Cavelan, V. Le Fèvre, Y. Robert, and H. Sun. “A different re-execution speed can help.” In: *International Conference on Parallel Processing Workshops (ICPPW)*. IEEE. 2016, pp. 250–257.

[W4] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. “Assessing General-Purpose Algorithms to Cope with Fail-Stop and Silent Errors.” In: *7th International Workshop in Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2014.

[W5] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. “Two-level checkpointing and verifications for linear task graphs.” In: *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2016, pp. 1239–1248.

[W6] A. Cavelan, Y. Robert, H. Sun, and F. Vivien. “Voltage overscaling algorithms for energy-efficient workflow computations with timing errors.” In: *Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. ACM. 2015, pp. 27–34.