N^o attribué par la bibliothèque : XXXXXXXX

- École normale supérieure de Lyon -
Laboratoire de l'Informatique du Parallélisme - UMR5668 - LIP

THÈSE

en vue d'obtenir le grade de

Docteur de l'École Normale Supérieure de Lyon
Spécialité : Informatique

au titre de l'École Doctorale Informatique et Mathématiques

présentée et soutenue publiquement le XXXX par

Dounia ZAIDOUNI

Combining checkpointing and other resilience mechanisms for exascale systems

Directeur de thèse :	Frédéric	VIVIEN
Co-encadrant de thèse :	Yves	ROBERT

Devant la commission d'examen formée de :

Olivier	Beaumont	*Rapporteur*
Denis	Trystram	*Rapporteur*
Christophe	Cérin	*Examinateur*
Jean-Marc	Nicod	*Examinateur*
Yves	Robert	*Co-encadrant*
Frédéric	Vivien	*Directeur de thèse*

Contents

Résumé

L'ensemble des contributions de cette thèse s'articule autour de problèmes d'ordonnancement et d'optimisation dans des contextes probabilistes. Ces contributions se déclinent en deux parties. La première partie est dédiée à l'optimisation de différents mécanismes de tolérance aux pannes pour les machines de très large échelle qui sont sujettes à une probabilité de pannes et la seconde partie est consacrée à l'optimisation du coût d'exécution des arbres d'opérateurs booléens sur des flux de données.

Dans la première partie, nous nous sommes intéressés aux problèmes de résilience pour les machines de future génération dites «exascales» (plateformes pouvant effectuer 10^{18} opérations par secondes).

Alors que la fiabilité des composants prise de manière indépendante augmente, le nombre de composants aussi augmente de manière exponentielle. Les quatre plateformes les plus puissantes dans la liste TOP 500 comprennent chacune plus de 500.000 coeurs et le temps moyen entre pannes (MTBF) d'une plateforme haute performance est inversement proportionnel à son nombre de processeurs. Ainsi, on s'attend à ce que les plateformes de future génération soient victimes en moyenne à plus d'une panne par jour et à ce que leur MTBF soit plus petit que le temps nécessaire pour faire une sauvegarde. D'où la nécessité d'avoir des mécanismes efficaces pour minimiser l'impact des pannes et pour les tolérer.

Le premier chapitre de cette partie est dédié à l'état de l'art des mécanismes les plus utilisés dans la tolérance aux pannes et à une présentation de résultats généraux liés à la résilience.

Dans le second chapitre, nous avons étudié un modèle d'évaluation des protocoles de sauvegarde de points de reprise (checkpoints) et de redémarrage. Le modèle proposé est suffisamment générique pour contenir les situations extrêmes: d'un côté le checkpoint coordonnée, et de l'autre toute une famille de stratégies non-coordonnées (avec enregistrement de messages). Nous avons identifié un ensemble de paramètres cruciaux pour l'instanciation et la comparaison de l'espérance de l'efficacité des protocoles de tolérance aux pannes, pour un couple donné application/plateforme. Nous avons proposé une analyse détaillée de plusieurs scénarios, incluant certaines des plateformes de calcul existantes les plus puissantes, ainsi que des anticipations sur les futures plateformes exascales.

Dans le troisième chapitre, nous avons étudié l'utilisation conjointe de la réplication et d'un mécanisme de checkpoints et de redémarrage. Avec la réplication, plusieurs processeurs exécutent le même calcul de telle sorte que la panne d'un processeur n'implique pas forcément une interruption de l'exécution de l'application. Dans ce chapitre nous avons considéré deux mises en œuvre de la réplication. Dans la première approche « Réplication de processus », chaque processus d'une unique instance d'une application parallèle est répliqué (de manière transparente). Dans la seconde approche «Réplication de l'application», des instances entières de l'application sont répliquées. La réplication de processus surpasse largement la réplication de l'application car la probabilité qu'un processus donné et ses instances répliquées tombent en panne est largement inférieure à la probabilité qu'une instance entière de l'application et ses instances répliquées tombent en panne. Cependant, la réplication de processus ne peut pas toujours être une option possible, car elle nécessite la modification de l'application. La réplication de l'application peut être utilisée chaque fois que la réplication de processus n'est pas possible car elle est agnostique du modèle de programmation parallèle, et donc elle considère l'application comme une boîte noire non modifiable. La seule exigence est que l'application soit malléable et que l'instance soit redémarrable à partir d'un fichier de point de sauvegarde. Concernant la première approche, nous avons dérivé de nouveaux résultats théoriques (Nombre Moyen de Pannes avant l'Échec et le Temps Moyen avant l'Échec) pour une distribution exponentielle des pannes. Nous avons étendu ces résultats à n'importe quel type de distribution, avec notamment des formules closes pour les distributions suivant des lois de Weibull. Concernant la seconde approche, nous avons fourni une étude théorique d'un schéma d'exécution avec réplication lorsque la distribution des pannes suit une loi exponentielle. Nous avons proposé des algorithmes de détermination des dates de sauvegarde quand la distribution des pannes suit une loi quelconque. Nous avons évalué les deux approches au moyen de simulations, basées sur une distribution de pannes suivant une loi exponentielle, de Weibull (ce qui est plus représentatif des systèmes réels), ou tirée de logs de clusters utilisés en production. Nos résultats montrent que la réplication est bénéfique pour un ensemble de modèles d'applications et de coût de sauvegardes réalistes, dans le cadre des futures plateformes exascales.

Dans le quatrième chapitre, nous avons étudié l'utilisation d'un prédicteur de pannes conjointement avec un mécanisme de checkpoints et de redémarrage. Un prédicteur de panne est un logiciel lié à une machine, qui par l'étude de «logs» (les événements enregistrés) et d'informations données par des capteurs sur la machine, va tenter de prédire le moment où une panne va arriver. Il est caractérisé par son taux de rappel (proportion des pannes effectivement prédites) et par sa précision (proportion de vraies pannes parmi toutes les pannes annoncées), et il fournit des prédictions soit exactes soit avec des intervalles de confiance. Dans ce chapitre, nous avons pu obtenir la valeur optimale de la période de checkpoint minimisant ainsi le gaspillage de l'utilisation des ressources dû au coût de prise de ces checkpoints et ce suivant différents scénarios. Ces résultats nous ont permis d'évaluer analytiquement les principaux paramètres qui influent sur la performance des prédicteurs de pannes à très grande échelle et de montrer que les paramètres les plus importants d'un prédicteur est son taux de rappel et non pas sa précision.

Dans le cinquième chapitre, nous avons considéré la technique traditionnelle de checkpoint et de redémarrage en présence de corruptions mémoires silencieuses. Contrairement aux pannes qui provoquent un arrêt de l'application, ces erreurs silencieuses ne sont pas détectées au moment où elles se produisent, mais plus tard, au moyen d'un mécanisme spécifique de détection. Dans ce chapitre nous avons considéré deux modèles, dans le premier modèle les erreurs sont détectées après un délai qui lui-même suit une distribution de probabilité (typiquement une loi exponentielle) et dans le deuxième modèle un appel à un mécanisme de vérification permet de détecter les erreurs lors de son invocation. Dans les deux cas nous sommes capables de calculer la période optimale minimisant les pertes, c'est-à-dire la partie du temps où les noeuds ne font pas de calculs utiles. En pratique, seul un nombre borné de checkpoints peut être gardé en

mémoire, et le premier modèle peut faire apparaître des fautes critiques qui provoquent la perte de tout le travail réalisé jusque là. Dans ce cas, nous avons calculé la période minimale qui satisfait une borne supérieure sur le risque. Pour le second modèle, il n'y a pas de risque de fautes critiques, grâce au mécanisme de vérification, mais le coût induit est reporté dans les pertes. Enfin, nous avons instancié et évalué chacun des modèles sous des scénarios et des paramètres d'architectures réalistes.

Dans la seconde partie de la thèse, nous avons étudié le problème de la minimisation du coût de récupération des données par des applications lors du traitement d'une requête exprimée sous forme d'arbres d'opérateurs booléens appliqués à des prédicats sur des flux de données de senseurs. Les données doivent être transférées des senseurs vers l'appareil de traitement des données, par exemple un smartphone. Transférer une donnée induit un coût, par exemple une consommation énergétique qui diminuera la charge de la batterie de l'agent mobile. Comme l'arbre de requêtes contient des opérateurs booléens, des pans de l'arbre peuvent être court-circuités en fonction des données récupérées. Un problème intéressant est de déterminer l'ordre dans lequel les prédicats doivent être évalués afin de minimiser l'espérance du coût du traitement de la requête. Ce problème a déjà été étudié sous l'hypothèse que chaque flux apparaît dans un seul prédicat.

Dans le sixième chapitre, nous avons présenté l'état de l'art de la seconde partie et dans le septième chapitre, nous avons étudié le problème de la minimisation du coût de récupération des données par des applications lors du traitement d'une requête exprimée sous forme normale disjonctive. Dans ce chapitre, nous avons éliminé l'hypothèse que chaque flux apparaît dans un seul prédicat et nous avons considéré le cas plus général où chaque flux peut apparaître dans plusieurs prédicats. Pour ce cas général, nous avons étudié deux modèles, le modèle où chaque prédicat peut accéder à un seul flux et le modèle où chaque prédicat peut accéder à plusieurs flux.

Pour le modèle où chaque prédicat peut accéder à un seul flux, nos principaux résultats sont un algorithme optimal pour les arbres avec un seul niveau, et une preuve de NP-complétude pour les arbres sous forme normale disjonctive. Cependant, pour les arbres sous forme normale disjonctive, nous avons montré qu'il existe un ordre optimal d'évaluation des prédicats qui correspond à un parcours en profondeur d'abord. Ce résultat nous a servi à concevoir toute une classe d'heuristiques. Nous avons montré que l'une de ces heuristiques a de bien meilleurs résultats que les autres heuristiques et a des résultats proches de l'optimale quand celui-ci peut-être calculer en temps raisonnable.

Pour le modèle où chaque prédicat peut accéder à plusieurs flux, on a proposé un algorithme glouton pour les arbres avec un seul niveau, cet algorithme est inspiré de l'algorithme optimal pour les arbres avec un seul niveau où chaque prédicat peut accéder à un seul flux. Nous avons présenté aussi une preuve de NP-complétude pour les arbres sous forme normale disjonctive et nous avons proposé aussi plusieurs heuristiques, les résultats de simulations ont montré qu'une heuristique a de bien meilleurs résultats que les autres heuristiques.

Introduction

As plans are made for deploying post-petascale high performance computing (HPC) systems [92, 95], solutions need to be developed to ensure that applications on such systems are resilient to faults. Resilience is particularly critical for large Petascale systems and future Exascale ones. These systems will typically gather from half a million to several millions of CPU cores. Table 1 presents the number of cores of the five fastest supercomputers which appeared at the top of the TOP500 list in June 2014 [1]. Future exascale supercomputers, which are expected by 2020, will be capable of at least one exaFLOPS (10^{18} floating point operations per second) which is 100 times the processing power of the fastest supercomputer currently operational, the Chinese Tianhe-2 supercomputer.

The mean-time between faults of a platform is inversely proportional to its number of components. For future generation platforms, processor failures are projected to be common occurrences [24, 25, 26], the mean-time between faults is expected to be so small that any application running for a non trivial duration on a whole platform will be the victim of at least one failure per day on average. For instance, the Jaguar platform had an average of 2.33 faults per day during the period from August 2008 to February 2010 [2]. So, fault-tolerance techniques become unavoidable for large-scale platforms.

Failures occur because not all faults are automatically detected and corrected in current production hardware. To tolerate failures, the standard protocol is *Coordinated Checkpointing* where all processors periodically stop computing and save the state of the parallel application onto resilient storage throughout execution, so that when a failure strikes some process, the entire application rolls back to a known consistent global state. More frequent checkpointing leads to higher overhead during fault-free execution, but less frequent checkpointing leads to a larger loss when a failure occurs. A checkpointing strategy specifies when checkpoints should be taken. A large literature is devoted to identifying good checkpointing strategies, including both theoretical and practical efforts. In spite of these efforts, the necessary checkpoint frequency for tolerating failures in large-scale platforms can become so high that processors spend more time checkpointing than computing. Consider an ideal moldable parallel application that can be executed on an arbitrary number of processors and that is perfectly parallel. The makespan

Supercomputers	**Cores**	**Location**
Tianhe-2 (MilkyWay-2) NUDT	3,120,000	Guangzhou, China
Titan - Cray XK7 Cray	560,640	Oak Ridge, USA
Sequoia - BlueGene/Q IBM	1,572,864	Livermore, USA
K computer Fujitsu	705,024	Kobe, Japan
Mira - BlueGene/Q IBM	786,432	Argonne, USA

Table 1: The five fastest supercomputers of TOP500 list in June 2014

with p processors is the sequential makespan divided by p. In a failure-free execution, the larger p the faster the execution. But in the presence of failures, as p increases so does the frequency of processor failures, leading to more time spent in recovering from these failures and leading to more time spent in more frequent checkpoints to allow for an efficient recovery after each failure. Beyond some threshold values, increasing p actually increases the expected makespan [24, 25, 26, 69]. This is because the MTBF (Mean Time Between Failures) of the platform becomes so small that the application performs too many recoveries and re-executions to make progress efficiently.

The main contributions of this thesis come in two parts. The common point between these two parts is that they both deal with scheduling and optimization problems in a probabilistic context. The first part of the thesis is devoted to the study of resilience on Exascale platforms. In this part we focus on the efficient execution of applications on failure-prone platforms. We typically address questions such as: Given a platform and an application, which fault-tolerance protocols should be used, when, and with which parameters? Here, we use probability distributions to describe the potential behavior of computing platforms, namely when hardware components are subject to faults. The second part is devoted to the optimization of the expected sensor data acquisition cost when evaluating a query expressed as a tree of disjunctive boolean operators applied to boolean predicates. Here, we evaluate a disjunctive normal form (DNF) tree assuming that we know the energy cost for retrieving each type of data, and the probability of success of each boolean operator.

The second part of this thesis is motivated by the fact that there is a growing interest in applications that use continuous sensing of individual activity via sensors embedded or associated with personal mobile devices. Solutions need to be developed to reduce the energy overheads of sensor data acquisition and processing and to ensure the successful continuous operation of such applications, especially on battery-limited mobile devices.

For instance, smartphones are equipped with increasingly sophisticated sensors (e.g., GPS, accelerometer, gyroscope, microphone) that enable near real-time sensing of an individual's activity or environmental context. A smartphone can then perform embedded query processing on the sensor data streams, e.g., for social networking [101], remote health monitoring [102]. The continuous processing of streams, even when data rates are moderate (such as for GPS or accelerometer data), can cause commercial smartphone batteries to be depleted in a few hours [103]. Hence the necessity to reduce the amount of sensor data acquired for mobile query processing devices.

Another example comes from automotive applications. These applications achieve their goals by continuously monitoring sensors in a vehicle, and fusing them with information from cloud databases, in order to detect events that are used to trigger actions (e.g., alerting a driver, turning on fog lights, screening calls). These sensors describe the instantaneous state and performance of many subsystems inside a vehicle, and represent a rich source of information, for assessing both vehicle behavior and driver behavior. For these applications, the queries describe specific rules of interest to a user, and the continuous processing of these queries requires the acquisition of the sensor data, which incurs a cost, e.g., energy cost due to byte transfers, latency, and bandwidth usage.

In this context, we consider query processing, when queries are modeled as a tree of disjunctive boolean operators, where the leaf nodes are the data sources and the root emits the result of the query. The evaluation of the query stops as soon as a truth value has been determined, possibly shortcircuiting part of the query tree. For instance, the result of an AND operator is known as soon as one of its inputs evaluates to false. Such a pruning reduces the amount of data to be retrieved and processed, and thus decreases the incurred costs of the query processing.

The objectif of our study is to reduce the incurred costs by minimizing the expected sensor data acquisition when evaluating the query. Each predicate is computed over data items from a particular data stream generated periodically by a sensor, and as a certain probability of evaluating to true. Some of the difficulties of the problem are that the leaf nodes may share some input data with other leaves or may use severals streams. These difficulties are well studied in the second part.

We now summarize the different chapters of this thesis briefly below.

Chapter 1: Foreword on resilience

This chapter discusses related work for Part I and introduces new general results for resilience on exascale systems.

Chapter 2: Unified Model for Assessing Checkpointing Protocols [J2]

In this chapter we present a unified model for several well-known checkpoint/restart protocols. The proposed model is generic enough to encompass both extremes of the checkpoint/restart space, from coordinated approaches to a variety of uncoordinated checkpoint strategies (with message logging). We identify a set of crucial parameters, instantiate them and compare the expected efficiency of the fault tolerant protocols, for a given application/platform pair. We then propose a detailed analysis of several scenarios, including some of the most powerful currently available HPC platforms, as well as anticipated Exascale designs. The results of this analytical comparison are corroborated by a comprehensive set of simulations. Altogether, they outline comparative behaviors of checkpoint strategies at very large scale, thereby providing insight that is hardly accessible to direct experimentation.

Chapter 3: Combining Replication and Coordinated Checkpointing [J3, RR6]

In this chapter, we study the mechanism of replication, which we used in addition to coordinated checkpointing. Using replication, multiple processors perform the same computation so that a processor failure does not necessarily mean application failure. While at first glance replication may seem wasteful, it may be significantly more efficient than using solely coordinated checkpointing at large scale. In this chapter we investigate two approaches for replication. In the first approach "Group replication", entire application instances are replicated. In the second approach "Process replication", each process in a single application instance is (transparently) replicated. Process replication largely outperforms group replication due to dramatically increased MTBF for each replica set. However, process replication may not always be a feasible option because it must be provided transparently as part of the runtime system. Group replication can be used whenever process replication is not available because it 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 moldable and that an instance be startable from a saved checkpoint file. In this chapter we provide a theoretical study of these two approaches, comparing them to the pure coordinated checkpointing approach in terms of expected application execution times.

Chapter 4: Combining Fault Prediction and Coordinated Checkpointing [J1, C3]

This chapter deals with the impact of fault prediction techniques on checkpointing strategies. In this chapter we deal with two problem instances, one where the predictor system provides "exact dates" for predicted events, and another where it only provides "prediction windows" during which events take place.

With a predictor with exact prediction dates, we extend the classical first-order analysis of Young and Daly in the presence of a fault prediction system, characterized by its recall and its precision. We provide optimal algorithms to decide whether and when to take predictions into account, and we derive the optimal value of the checkpointing period. These results allow to analytically assess the key parameters that impact the performance of fault predictors at very large scale.

With a predictor with a prediction window, the analysis of the checkpointing strategies is more complicated. We propose a new approach based upon two periodic modes, a regular mode outside prediction windows, and a proactive mode inside prediction windows, whenever the size of these windows is large enough. We are able to compute the best period for any size of the prediction windows, thereby deriving the scheduling strategy that minimizes platform waste. In addition, the results of the analytical study are nicely corroborated by a comprehensive set of simulations, which demonstrates the validity of the model and the accuracy of the approach.

Chapter 5: Combining Silent Error Detection and Coordinated Checkpointing [C2]

In this chapter, we revisit traditional checkpointing and rollback recovery strategies, with a focus on silent data corruption errors. Contrarily to fail-stop failures, such latent errors cannot be detected immediately, and a mechanism to detect them must be provided. We consider two models: (i) errors are detected after some delays following a probability distribution (typically, an Exponential distribution); (ii) errors are detected through some verification mechanism. In both cases, we compute the optimal period in order to minimize the waste, i.e., the fraction of time where nodes do not perform useful computations. In practice, only a fixed number of checkpoints can be kept in memory, and the first model may lead to an irrecoverable failure. In this case, we compute the minimum period required for an acceptable risk. For the second model, there is no risk of irrecoverable failure, owing to the verification mechanism, but the corresponding overhead is included in the waste. Finally, both models are instantiated using realistic scenarios and application/architecture parameters.

Chapter 6: Related Work

This chapter discusses in details the related work for Part II.

Chapter 7: Cost-Optimal Execution of Boolean DNF Trees with Shared Streams [C1, RR1]

The objective of this chapter is to determine the order in which predicates should be evaluated so as to shortcut part of the query evaluation and minimize the expected cost. This problem has been studied assuming that each data stream occurs at a single predicate. In this chapter we remove this assumption since it does not necessarily hold in practice. We study two cases in which a stream can occur in multiple leaves. In the first case, we consider Boolean DNF Trees with leaves accessing a single stream and in the second case, we considered Boolean

DNF Trees with leaves accessing multiple streams. We term the first case *single-stream* case and the second case *multi-stream* case.

For AND query trees, we give a polynomial-time greedy algorithm that is optimal in the *single-stream* case, and show that the problem in NP-complete in the *multi-stream* case. For the *multi-stream* case we propose an extension of the *single-stream* greedy algorithm. This extension is not optimal but computes near-optimal leaf evaluation orders in practice.

For DNF query trees, we show that the problem is NP-complete in the *single-stream* case (and thus also in the *multi-stream* case). In both the *single-stream* and *multi-stream* case we show that there exists an optimal leaf evaluation order that is depth-first. This result provides inspiration for a class of heuristics that we evaluate in simulation and compare to the optimal solution (computed via an exhaustive search on small instances). We show that one of these heuristics largely outperforms other sensible heuristics, including a heuristic proposed in previous work.

Part I

Resilience on exascale systems

Chapter 1

Foreword on resilience

In this chapter, we start by discussing the related work for Part I, and then we introduce some general results that are useful for resilience on exascale systems. The first of these results is relative to the MTBF μ of a platform made of N individual components whose individual MTBF is μ_{ind}: $\mu = \frac{\mu_{\text{ind}}}{N}$. This result is widely used, but to the best of our knowledge, no proof has ever been published before. The second result is a refined first-order analysis for instantaneous fault detection. When faults follow an exponential distribution, it leads to similar periods as those obtained by Young [53] and Daly [54], but leads to better performance when faults follow a Weibull distribution.

1.1 Related work

The standard fault tolerance protocol is *Coordinated Checkpointing*. It has been widely studied in the literature. The major appeal of the coordinated checkpointing protocol is its simplicity, because a parallel job using n processors of individual MTBF μ_{ind} can be viewed as a single processor job with MTBF $\mu = \frac{\mu_{ind}}{n}$ (as will be shown in Section 1.2.1). Given the value of μ, an approximation of the optimal checkpointing period can be computed as a function of the key parameters (downtime D, checkpoint time C, and recovery time R).

As already mentioned, the first estimate had been given by Young [53] and later refined by Daly [54]. Both use a first-order approximation for Exponential failure distributions. Daly extended his work in [30] to study the impact of sub-optimal checkpointing periods. In [41], the authors develop an "optimal" checkpointing policy, based on the popular assumption that optimal checkpointing must be periodic. In [77], Bouguerra et al. prove that the optimal checkpointing policy is periodic when checkpointing and recovery overheads are constant, for either Exponential or Weibull failures. But their results rely on the unstated assumption that all processors are rejuvenated after each failure and after each checkpoint. In [61], the authors have shown that this assumption is unreasonable for Weibull failure and they have developed optimal solutions for Exponential failure distributions. Dynamic programming heuristics for arbitrary distributions are proposed in [78, 79, 61].

The literature proposes different works [80, 81, 82, 83, 84] on the modeling of coordinated checkpointing protocols. In particular, [81] and [80] focus on the usage of available resources: some may be kept as backup in order to replace the down ones, and others may be even shutdown in order to decrease the failure risk or to prevent storage consumption by saving fewer checkpoint snapshots.

The major drawback of coordinated checkpointing protocols is their lack of scalability at extreme-scale. These protocols will lead to I/O congestion when too many processes are checkpointing at the same time. Even worse, transferring the whole memory footprint of an HPC application onto stable storage may well take so much time that a failure is likely to take place during the transfer! A few papers [84, 68] propose a scalability study to assess the impact of a small MTBF (i.e., of a large number of processors).

The mere conclusion is that checkpoint time should be either dramatically reduced for platform waste to become acceptable, or that checkpointing should be coupled with other fault tolerant mechanisms. The latter option has motivated the chapters of the first part of this thesis.

We present hereafter the related work of each chapter of Part I.

Models for assessing checkpointing protocols. There are two approaches of fault tolerant protocols; On one extreme, *coordinating checkpoints*, where after a failure, the entire application rolls back to a known consistent global state; On the opposite extreme, *message logging*, which allows for independent restart of failed processes but logs supplementary state elements during the execution to drive a directed replay of the recovering processes. The interested reader can refer to [18] for a comprehensive survey of message logging approaches, and to [96] for a description of the most common algorithm for checkpoint coordination.

Although the uncoordinated nature of the restart in message logging improves recovery speed compared to the coordinated approach (during the replay, all incoming messages are available without jitter, most emissions are discarded and other processes can continue their progress until they need to synchronize with replaying processes) [16], the logging of message payload incurs a communication overhead and an increase in the size of checkpoints directly influenced by the communication intensity of the application [19]. Recent advances in message logging [20, 22, 21] have led to composite algorithms, called *hierarchical checkpointing*, capable of partial coordination of checkpoints to decrease the cost of logging, while retaining message logging capabilities to remove the need for a global restart.

These hierarchical schemes partition the application processes into groups. Each group checkpoints independently, but processes belonging to the same group coordinate their checkpoints and recovery. Communications between groups continue to incur payload logging. However, because processes belonging to a same group follow a coordinated checkpointing protocol, the payload of messages exchanged between processes within the same group is not required to be logged.

The optimizations driving the choice of the size and shape of groups are varied. A simple heuristic is to checkpoint as many processes as possible, simultaneously, without exceeding the capacity of the I/O system. In this case, groups do not checkpoint in parallel. Groups can also be formed according to hardware proximity or communication patterns. In such approaches, there may be opportunity for several groups to checkpoint concurrently. The model that we propose in Chapter 2 captures the intricacies of all such strategies, thereby also representing both extremes, coordinated and uncoordinated checkpointing. In Section 2.3, we describe the meaningful parameters to instantiate these various protocols for a variety of platforms and applications, taking into account the overhead of message logging, and the impact of grouping strategies.

The question of the optimal checkpointing period for sequential jobs (or parallel jobs undergoing coordinated checkpointing) has seen many studies presenting different order of estimates: see [53, 54], and [75, 76] that consider Weibull distributions, or [14] that considers parallel jobs. This is critical to extract the best performance of any rollback-recovery protocol. However in Chapter 2, although we use the same approach to find the optimal checkpoint interval, we focus our study on the comparison of different protocols that were not captured by previous models.

The literature proposes different works [80, 81, 82, 83, 84] on the modeling of coordinated checkpointing protocols. [79] focuses on refining failure prediction; [81] and [80] focus on the usage of available resources. [84] proposes a scalability model where they evaluate the impact of failures on application performance. A significant difference with these works lies in the inclusion of several new parameters to refine the model.

The uncoordinated and hierarchical checkpointing have been less frequently modeled. [15] models *periodic* checkpointing on *fault-aware* parallel tasks that do not communicate. From our point of view, this specificity does not match the uncoordinated checkpointing with message logging that we consider. In Chapter 2, we consistently address all the three families of checkpointing protocols: coordinated, uncoordinated, and hierarchical ones. To the best of our knowledge, it is the first attempt at providing a unified model for this large spectrum of protocols.

Replication. Replication has long been used as a fault-tolerance mechanism in distributed systems [42]. The idea to use replication together with checkpoint-recovery has been studied in the context of grid computing [31]. One concern about replication in HPC is the induced resource waste. However, given the scalability limitations of pure checkpoint-recovery, replication has recently received more attention in the HPC literature [26, 33, 32].

In Chapter 3 we study two replication techniques, group replication and process replication. While, to the best of our knowledge, no previous work has considered group replication, process replication has been studied by several authors. Process replication is advocated in [34] for HPC applications, and in [35] for grid computing with volatile nodes. The work by Ferreira et al. [69] has studied the use of process replication for MPI (Message Passing Interface) 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 [36, 37] (in combination with coordinated checkpointing) to decrease the overhead associated to full replication. Adaptive redundancy is introduced in [38], where a subset of processes is dynamically selected for replication.

In Section 3.4, we provide a full-fledge theoretical analysis of the combination of process replication and checkpoint-recovery. While some theoretical results are provided in [69], they are based on an analogy between the process replication problem and the birthday problem. This analogy is appealing but, as seen in Section 3.4.1, does not make it possible to compute exact *MNFTI* and *MTTI* values. In addition, the authors of [69] use Daly's formula for the checkpointing period, even for Weibull or other distributions, simply using the mean of the distribution in the formula. This is a commonplace approach. However, a key observation is that using replication changes the optimal checkpointing period, even for Exponential distributions. Chapter 3 provides the optimal value of the period for Exponential and Weibull distributions (either analytically or experimentally), taking into account the use of replication.

Fault Prediction. Considerable research has been devoted to fault prediction, using very different models (system log analysis [51], event-driven approach [48, 51, 52], support vector machines [50, 47], nearest neighbors [50], etc). We give a brief overview of existing predictors, focusing on their characteristics rather than on the methods of prediction. For the sake of

Paper	Lead Time	Precision	Recall	Prediction Window
[52]	300 s	40 %	70%	-
[52]	600 s	35 %	60%	-
[51]	2h	64.8 %	65.2%	yes (size unknown)
[51]	0 min	82.3 %	85.4 %	yes (size unknown)
[48]	32 s	93 %	43 %	-
[49]	10s	92 %	40 %	-
[49]	60s	92 %	20 %	-
[49]	600s	92 %	3 %	-
[47]	NA	70 %	75 %	-
[50]	NA	20 %	30 %	1h
[50]	NA	30 %	75 %	4h
[50]	NA	40 %	90 %	6h
[50]	NA	50 %	30 %	6h
[50]	NA	60 %	85%	12h

Table 1.1: Comparative study of different parameters returned by some predictors.

clarity, we sum up the characteristics of the different fault predictors from the literature in Table 1.1.

A predictor is characterized by two critical parameters, its recall r, which is the fraction of faults that are indeed predicted, and its precision p, which is the fraction of predictions that are correct (i.e., correspond to actual faults).

The authors of [52] introduce the *lead time*, that is the duration between the time the prediction is made and the time the predicted fault is supposed to happen. This time should be sufficiently large to enable proactive actions. The distribution of lead times is irrelevant. Indeed, only predictions whose lead time is greater than C_p, the time to take a proactive checkpoint, are meaningful. Predictions whose lead time is smaller than C_p, whenever they materialize as actual faults, should be classified as unpredicted faults; the predictor recall should be decreased accordingly.

The predictor of [52] is also able to locate where the predicted fault is supposed to strike. This additional characteristics has a negative impact on the precision (because a fault happening at the predicted time but not on the predicted location is classified as a non predicted fault; see the low value of p in Table 1.1). The authors of [52] state that fault localization has a positive impact on proactive checkpointing time in their context: instead of a full checkpoint costing $1,500$ seconds, they can take a partial checkpoint costing only 12 seconds. This led us to introduce a different cost C_p for proactive checkpoints, which can be smaller than the cost C of regular checkpoints. Gainaru et al. [49] also stated that fault-localization could help decrease the checkpointing time. Their predictor also gives information on fault localization. They studied the impact of different lead times on the recall of their predictor.

The authors of [51] also consider a lead time, and introduce a *prediction window* indicating when the predicted fault should happen. The authors of [50] study the impact of different prediction techniques with different prediction window sizes. They also consider a lead time, but do not state its value. These two latter studies motivate the work of Section 4.3, even though [51] does not provide the size of their prediction window.

Most studies on fault prediction state that a proactive action must be taken right before the predicted fault, be it a checkpoint or a migration. However, we show in Section 4.2 that it is beneficial to ignore some predictions, namely when the predicted fault is announced to strike less than $\frac{C_p}{p}$ seconds after the last periodic checkpoint.

Unfortunately, much of the work done on prediction does not provide information that could be really useful for the design of efficient algorithms. Missing information includes the lead time and the size of the prediction window. Other information that could be useful would be: (i) the distribution of the faults in the prediction window; and (ii) the precision and recall as functions of the size of the prediction window (what happens with a larger prediction window). In the simpler case where predictions are exact-date predictions, Gainaru et al [49] and Bouguerra et al. [71] have shown that the optimal checkpointing period becomes $T_{\text{opt}} = \sqrt{\frac{2\mu C}{1-r}}$, but their analysis is valid only if μ is very large in front of the other parameters and their computation of the waste is not fully accurate [RR2]. In Section 4.2, we have refined the results of [49], focusing on a more accurate analysis of fault prediction with exact dates, and providing a detailed study on the impact of recall and precision on the waste. As shown in Subsection 4.3.2, the analysis of the waste is dramatically more complicated when using prediction windows than when using exact-date predictions.

Li et al. [70] considered the mathematical problem of when and how to migrate. In order to be able to use migration, they assumed that at any time 2% of the resources are available as spares. This allows them to conceive a Knapsack-based heuristic. Thanks to their algorithm, they were able to save 30% of the execution time compared to a heuristic that does not take the prediction into account, with a precision and recall of 70%, and with a maximum load of 0.7. In our study, we do not consider that we have a batch of spare resources. We assume that after a downtime the resources that failed are once again available.

To the best of our knowledge, this work is the first to focus on the mathematical aspect of fault prediction, and to provide a model and a detailed analysis of the waste due to all three types of events (true and false predictions and unpredicted failures).

Silent Error Detection. All the above 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. These schemes assume instantaneous error detection (therefore mainly targeting fail-stop failures) and are unable to accommodate silent errors.

Considerable efforts have been directed at error-checking to reveal latent 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 [73]. For high-performance scientific applications, process replication (each process is equipped with a replica, and messages are quadruplicated) is proposed in the RedMPI library [89]. Application-specific information can be very useful to enable ad-hoc solutions, that dramatically decrease the cost of detection. Many techniques have been advocated. They include memory scrubbing [88], but also ABFT techniques [90, 87, 3], such as coding for the sparse-matrix vector multiplication kernel [3], and coupling a higher-order with a lower-order scheme for PDEs [4]. 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 in [5]. See also [86] for the design of a fault-tolerant GMRES capable of converging despite silent errors, and [85] for a comparative study of detection costs for iterative methods. Lu, Zheng and Chien [74] give a comprehensive list of techniques and references. In Chapter 5, our work is agnostic of the underlying error-detection technique and takes the cost of verification as an input parameter to the model (see Section 5.3).

1.2 General results

1.2.1 MTBF of a platform

When considering a platform prone to failures, the key parameter is μ, the MTBF of the platform. If the platform is made of N components whose individual MTBF is μ_{ind}, then $\mu = \frac{\mu_{\text{ind}}}{N}$. This result is true regardless of the fault distribution law.

Proposition 1. *Consider a platform comprising N components, and assume that the inter-arrival times of the faults on the components are independent and identically distributed random variables that follow an arbitrary probability law whose expectation is finite and $\mu_{ind} > 0$. Then the MTBF μ of the platform (whose inverse is defined as the expectation of the sum of failure numbers of the N processors over time), is equal to $\frac{\mu_{ind}}{N}$.*

Proof. Consider first a single component, say component number q. Let X_i, $i \geq 0$ denote the IID random variables for fault inter-arrival times on that component, with $\mathbb{E}(X_i) = \mu_{\text{ind}}$. Consider a fixed time bound F. Let $n_q(F)$ be the number of faults on the component until time F is exceeded. In other words, the $(n_q(F) - 1)$-th fault is the last one to happen strictly before time F, and the $n_q(F)$-th fault is the first to happen at time F or after. By definition of $n_q(F)$, we have

$$\sum_{i=1}^{n_q(F)-1} X_i \leq F \leq \sum_{i=1}^{n_q(F)} X_i$$

Using Wald's equation [58, p. 486], with $n_q(F)$ as a stopping criterion, we derive:

$$(\mathbb{E}(n_q(F)) - 1)\mu_{\text{ind}} \leq F \leq \mathbb{E}(n_q(F))\,\mu_{\text{ind}}$$

and we obtain:

$$\lim_{F\to+\infty} \frac{\mathbb{E}(n_q(F))}{F} = \frac{1}{\mu_{\text{ind}}} \tag{1.1}$$

Consider now the whole platform, and let Y_i, $i \geq 0$ denote the IID random variables for fault inter-arrival times on the platform, with $\mathbb{E}(Y_i) = \mu$. Consider a fixed time bound F as before. Let $n(F)$ be the number of faults on the whole platform until time F is exceeded. Let $m_q(F)$ be the number of these faults that strike component number q. Of course we have $n(F) = \sum_{q=1}^{N} m_q(F)$. By definition, except for the component hit by the last failure, $m_q(F) + 1$ is the number of failures on component q until time F is exceeded, hence $n_q(F) = m_q(F)+1$ (and this number is $m_q(F) = n_q(F)$ on the component hit by the last failure). From Equation (1.1) again, we have for each component q:

$$\lim_{F\to+\infty} \frac{\mathbb{E}(m_q(F))}{F} = \frac{1}{\mu_{\text{ind}}}$$

Since $n(F) = \sum_{q=1}^{N} m_q(F)$, we also have:

$$\lim_{F\to+\infty} \frac{\mathbb{E}(n(F))}{F} = \frac{N}{\mu_{\text{ind}}} \tag{1.2}$$

■

1.2.2 Revisiting Daly's first-order approximation

Young proposed in [53] a "first order approximation to the optimum checkpoint interval". Young's formula was later refined by Daly [54] to take into account the recovery time. We revisit their analysis using the notion of waste. We recall that in the following, C is the time to execute a checkpoint, D the duration of a downtime, and R the duration of the recovery of a checkpoint (following a downtime).

Let $\textsc{Time}_{\text{base}}$ be the base time of the application without any overhead (neither checkpoints nor faults). First, assume a *fault-free* execution of the application with periodic checkpointing. In such an environment, during each period of length T we take a checkpoint, which lasts for a time C, and only $T-C$ units of work are executed. Let $\textsc{Time}_{\text{FF}}$ be the execution time of the application in this setting. Following most works in the literature, we also take a checkpoint at the end of the execution. The fault-free execution time $\textsc{Time}_{\text{FF}}$ is equal to the time needed to execute the whole application, $\textsc{Time}_{\text{base}}$, plus the time taken by the checkpoints:

$$\textsc{Time}_{\text{FF}} = \textsc{Time}_{\text{base}} + N_{\text{ckpt}} C \tag{1.3}$$

where N_{ckpt} is the number of checkpoints taken. We have

$$N_{\text{ckpt}} = \left\lceil \frac{\textsc{Time}_{\text{base}}}{T-C} \right\rceil \approx \frac{\textsc{Time}_{\text{base}}}{T-C}$$

When discarding the ceiling function, we assume that the execution time is very large with respect to the period or, symmetrically, that there are many periods during the execution. Plugging back the (approximated) value $N_{\text{ckpt}} = \frac{\textsc{Time}_{\text{base}}}{T-C}$, we derive that

$$\textsc{Time}_{\text{FF}} = \frac{\textsc{Time}_{\text{base}}}{T-C} T \tag{1.4}$$

The waste due to checkpointing in a fault-free execution, $\textsc{Waste}_{\text{FF}}$, is defined as the fraction of the execution time that does not contribute to the progress of the application:

$$\textsc{Waste}_{\text{FF}} = \frac{\textsc{Time}_{\text{FF}} - \textsc{Time}_{\text{base}}}{\textsc{Time}_{\text{FF}}} \quad \Leftrightarrow \quad (1-\textsc{Waste}_{\text{FF}})\textsc{Time}_{\text{FF}} = \textsc{Time}_{\text{base}} \tag{1.5}$$

Combining Equations (1.4) and (1.5), we get:

$$\textsc{Waste}_{\text{FF}} = \frac{C}{T} \tag{1.6}$$

Now, let $\textsc{Time}_{\text{final}}$ denote the expected execution time of the application in the presence of faults. This execution time can be divided into two parts: (i) the execution of "chunks" of work of size $T-C$ followed by their checkpoint; and (ii) the time lost due to the faults. This decomposition is illustrated by Figure 1.1. The first part of the execution time is equal to $\textsc{Time}_{\text{FF}}$. Let N_{faults} be the number of faults occurring during the execution, and let T_{lost} be the average time lost per fault. Then,

$$\textsc{Time}_{\text{final}} = \textsc{Time}_{\text{FF}} + N_{\text{faults}} \times T_{\text{lost}} \tag{1.7}$$

On average, during a time $\textsc{Time}_{\text{final}}$, $N_{\text{faults}} = \frac{\textsc{Time}_{\text{final}}}{\mu}$ faults happen. We need to estimate T_{lost}. The instants at which periods begin and at which faults strike can be considered independent, as a first order approximation. Therefore, the expected time elapsed between the completion of the last checkpoint and a fault is approximated as $\frac{T}{2}$ for all distribution laws, regardless of their particular shape. This approximation has been proven exact by Daly [54] for exponential laws, and we use it in the general case. We conclude that $T_{\text{lost}} = \frac{T}{2} + D + R$, because after each fault there is a downtime and a recovery. This leads to:

$$\textsc{Time}_{\text{final}} = \textsc{Time}_{\text{FF}} + \frac{\textsc{Time}_{\text{final}}}{\mu} \times \left(D + R + \frac{T}{2}\right)$$

Let $\textsc{Waste}_{\text{fault}}$ be the fraction of the total execution time that is lost because of faults:

$$\textsc{Waste}_{\text{fault}} = \frac{\textsc{Time}_{\text{final}} - \textsc{Time}_{\text{FF}}}{\textsc{Time}_{\text{final}}} \quad \Leftrightarrow \quad (1 - \textsc{Waste}_{\text{fault}})\,\textsc{Time}_{\text{final}} = \textsc{Time}_{\text{FF}} \tag{1.8}$$

We derive:

$$\textsc{Waste}_{\text{fault}} = \frac{1}{\mu}\left(D + R + \frac{T}{2}\right). \tag{1.9}$$

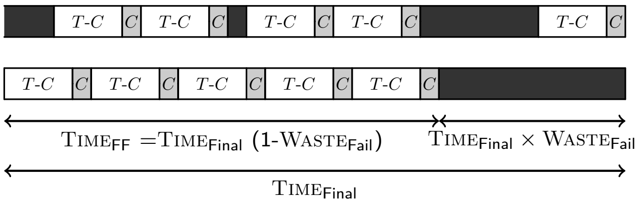


Figure 1.1: An execution (top), and its re-ordering (bottom), to illustrate both sources of waste. Blackened intervals correspond to work destroyed by faults, downtimes, and recoveries.

In [54], Daly uses the expression

$$\textsc{Time}_{\text{final}} = (1 + \textsc{Waste}_{\text{fault}})\textsc{Time}_{\text{FF}} \tag{1.10}$$

instead of Equation (1.8), which leads him to his well-known first-order formula

$$T = \sqrt{2(\mu + (D + R))C} + C \tag{1.11}$$

Figure 1.1 explains why Equation (1.10) is not correct and should be replaced by Equation (1.8). Indeed, the expected number of faults depends on the final time, not on the time for a fault-free execution. We point out that Young [53] also used Equation (1.10), but with $D = R = 0$. Equation (1.8) can be rewritten $\textsc{Time}_{\text{final}} = \textsc{Time}_{\text{FF}}/(1 - \textsc{Waste}_{\text{fault}})$. Therefore, using Equation (1.10) instead of Equation (1.8), in fact, is equivalent to write $\frac{1}{1-\textsc{Waste}_{\text{fault}}} \approx 1 + \textsc{Waste}_{\text{fault}}$ which is indeed a first-order approximation if $\textsc{Waste}_{\text{fault}} \ll 1$.

Now, let $\textsc{Waste}$ denote the total waste:

$$\textsc{Waste} = \frac{\textsc{Time}_{\text{final}} - \textsc{Time}_{\text{base}}}{\textsc{Time}_{\text{final}}} \tag{1.12}$$

Therefore

$$\textsc{Waste} = 1 - \frac{\textsc{Time}_{\text{base}}}{\textsc{Time}_{\text{final}}} = 1 - \frac{\textsc{Time}_{\text{base}}}{\textsc{Time}_{\text{FF}}}\frac{\textsc{Time}_{\text{FF}}}{\textsc{Time}_{\text{final}}} = 1 - (1 - \textsc{Waste}_{\text{FF}})(1 - \textsc{Waste}_{\text{fault}}).$$

Altogether, we derive the final result:

$$\begin{aligned}\textsc{Waste} &= \textsc{Waste}_{\text{FF}} + \textsc{Waste}_{\text{fault}} - \textsc{Waste}_{\text{FF}}\textsc{Waste}_{\text{fault}} && (1.13)\\ &= \frac{C}{T} + \left(1 - \frac{C}{T}\right)\frac{1}{\mu}\left(D + R + \frac{T}{2}\right) && (1.14)\end{aligned}$$

We obtain $\textsc{Waste} = \frac{u}{T} + v + wT$ where $u = C(1 - \frac{D+R}{\mu})$, $v = \frac{D+R-C/2}{\mu}$, and $w = \frac{1}{2\mu}$. Thus $\textsc{Waste}$ is minimized for $T = \sqrt{\frac{u}{w}}$. The Refined First-Order (RFO) formula for the optimal period is thus:

$$T_{\text{RFO}} = \sqrt{2(\mu - (D + R))C} \quad (1.15)$$

It is interesting to point out why Equation (1.15) is a first-order approximation, even for large jobs. Indeed, there are several restrictions to enforce for the approach to be valid:

— We have stated that the expected number of faults during execution is $N_{\text{faults}} = \frac{\textsc{Time}_{\text{final}}}{\mu}$, and that the expected time lost due to a fault is $T_{\text{lost}} = \frac{T}{2}$. Both statements are true individually, but the expectation of a product is the product of the expectations only if the random variables are independent, which is not the case here because $\textsc{Time}_{\text{final}}$ depends upon the failure inter-arrival times.

— We have used that the computation time lost when a failure happens is $\frac{T}{2}$ which has been proven true for exponential and uniform distributions only.

— In Equation (1.6), we have to enforce $C \leq T$ to have $\textsc{Waste}_{\text{FF}} \leq 1$.

— In Equation (1.9), we have to enforce $D + R \leq \mu$ and to bound T in order to have $\textsc{Waste}_{\text{fault}} \leq 1$. Intuitively, we need μ to be large enough for Equation (1.9) to make sense. However, regardless of the value of the individual MTBF μ_{ind}, there is always a threshold in the number of components N above which the platform MTBF $\mu = \frac{\mu_{\text{ind}}}{N}$ becomes too small for Equation (1.9) to be valid.

— Equation (1.9) is accurate only when two or more faults do not take place within the same period. Although unlikely when μ is large in front of T, the possible occurrence of many faults during the same period cannot be eliminated.

To ensure that the latter condition (at most a single fault per period) is met with a high probability, we cap the length of the period: we enforce the condition $T \leq \alpha\mu$, where α is some tuning parameter chosen as follows. The number of faults during a period of length T can be modeled as a Poisson process of parameter $\beta = \frac{T}{\mu}$. The probability of having $k \geq 0$ faults is $P(X = k) = \frac{\beta^k}{k!}e^{-\beta}$, where X is the number of faults. Hence the probability of having two or more faults is $\pi = P(X \geq 2) = 1 - (P(X = 0) + P(X = 1)) = 1 - (1 + \beta)e^{-\beta}$. If we assume $\alpha = 0.27$ then $\pi \leq 0.03$, hence a valid approximation when bounding the period range accordingly. Indeed, with such a conservative value for α, we have overlapping faults for only 3% of the checkpointing segments in average, so that the model is quite reliable. For consistency, we also enforce the same type of bound on the checkpoint time, and on the downtime and recovery: $C \leq \alpha\mu$ and $D + R \leq \alpha\mu$. However, enforcing these constraints may lead to use a sub-optimal period: it may well be the case that the optimal period $\sqrt{2(\mu - (D + R))C}$ of Equation (1.15) does not belong to the admissible interval $[C, \alpha\mu]$. In that case, the waste is minimized for one of the bounds of the admissible interval: this is because, as seen from Equation (1.14), the waste is a convex function of the period.

We conclude this discussion on a positive note. While capping the period, and enforcing a lower bound on the MTBF, is mandatory for mathematical rigor, simulations (see Chapter 4.2 for both Exponential and Weibull distributions) show that actual job executions can always use the value from Equation (1.15), accounting for multiple faults whenever they occur by re-executing the work until success. The first-order model turns out to be surprisingly robust!

To the best of our knowledge, despite all the limitations above, there is no better approach to estimate the waste due to checkpointing when dealing with arbitrary fault distributions. However, assuming that faults obey an Exponential distribution, it is possible to use the memory-less property of this distribution to provide more accurate results. A second-order approximation when faults obey an Exponential distribution is given in Daly [54, Equation (20)] as $\textsc{Time}_{\text{final}} = \mu e^{R/\mu}(e^{\frac{T}{\mu}} - 1)\frac{\textsc{Time}_{\text{base}}}{T-C}$. In fact, in that case, the exact value of $\textsc{Time}_{\text{final}}$ is provided in [61, 62] as $\textsc{Time}_{\text{final}} = (\mu + D)e^{R/\mu}(e^{\frac{T}{\mu}} - 1)\frac{\textsc{Time}_{\text{base}}}{T-C}$, and the optimal period is then $\frac{1+\mathbb{L}(-e^{-\frac{C}{\mu}-1})}{\mu}$ where $\mathbb{L}$, the Lambert function, is defined as $\mathbb{L}(z)e^{\mathbb{L}(z)} = z$.

To assess the accuracy of the different first order approximations, we compare the periods defined by Young's formula [53], Daly's formula [54], and Equation (1.15), to the optimal period, in the case of an Exponential distribution. Results are reported in Table 1.2. To establish these results, we use the same parameters as in Section 4.2.3: $C = R = 600$ s, $D = 60$ s, and $\mu_{\text{ind}} = 125$ years. One can observe in Table 1.2 that the relative error for Daly's period is slightly larger than the one for Young's period. In turn, the absolute value of the relative error for Young's period is slightly larger than the one for RFO. More importantly, when Young's and Daly's formulas overestimate the period, RFO underestimates it. Table 1.2 does not allow us to assess whether these differences are actually significant. However we also report in Chapter 4.2 some simulations that show that Equation (1.15) leads to smaller execution times for Weibull distributions than both classical formulas (Tables 4.3 and 4.4).

N	μ	Young		Daly		RFO		Optimal
2^{10}	3849609	68567	(0.5 %)	68573	(0.5 %)	67961	(-0.4 %)	68240
2^{11}	1924805	48660	(0.7 %)	48668	(0.7 %)	48052	(-0.6 %)	48320
2^{12}	962402	34584	(1.2 %)	34595	(1.2 %)	33972	(-0.6 %)	34189
2^{13}	481201	24630	(1.6 %)	24646	(1.7 %)	24014	(-0.9 %)	24231
2^{14}	240601	17592	(2.3 %)	17615	(2.5 %)	16968	(-1.3 %)	17194
2^{15}	120300	12615	(3.2 %)	12648	(3.5 %)	11982	(-1.9 %)	12218
2^{16}	60150	9096	(4.5 %)	9142	(5.1 %)	8449	(-2.9 %)	8701
2^{17}	30075	6608	(6.3 %)	6673	(7.4 %)	5941	(-4.4 %)	6214
2^{18}	15038	4848	(8.8 %)	4940	(10.8 %)	4154	(-6.8 %)	4458
2^{19}	7519	3604	(12.0 %)	3733	(16.0 %)	2869	(-10.8 %)	3218

Table 1.2: Comparing periods produced by the different approximations with optimal value. Beside each period, we report its relative deviation to the optimal. Each value is expressed in seconds.

Chapter 2

Unified Model for Assessing Checkpointing Protocols

2.1 Introduction

A significant research effort is focusing on the characteristics, features, and challenges of High Performance Computing (HPC) systems capable of reaching the Exaflop performance mark [92, 95]. The portrayed Exascale systems will necessitate billion way parallelism, resulting not only in a massive increase in the number of processing units (cores), but also in terms of computing nodes.

Considering the relative slopes describing the evolution of the reliability of individual components on one side, and the evolution of the number of components on the other side, the reliability of the entire platform is expected to decrease, due to probabilistic amplification. Executions of large parallel applications on these systems will have to tolerate a higher degree of errors and failures than in current systems. Preparation studies forecast that standard fault tolerance approaches (*e.g.*, coordinated checkpointing on parallel file system) will lead to unacceptable overheads at Exascale. Thus, it is not surprising that improving fault tolerance techniques is one of the main recommendations isolated by these studies [92, 95].

In this chapter we focus on techniques tolerating the effect of detected errors that prevent successful completion of the application execution. Undetected errors, also known as silent errors, are out-of-scope of this analysis. There are two main ways of tolerating process crashes, without undergoing significant application code refactoring: replication and rollback recovery. An analysis of replication feasibility for Exascale systems was presented in [69]. In this chapter we focus on rollback recovery, and more precisely on the comparison of checkpointing protocols.

There are three main families of checkpointing protocols: (i) coordinated checkpointing; (ii) uncoordinated checkpointing with message logging; and (iii) hierarchical protocols mixing coordinated checkpointing and message logging. The key principle in all these checkpointing protocols is that all data and states necessary to restart the execution are regularly saved in process *checkpoints*. Depending on the protocol, these checkpoints are or are not guaranteed to form consistent recovery lines. When a failure occurs, appropriate processes *rollback* to their last checkpoints and resume execution.

Each protocol family has serious drawbacks. Coordinated checkpointing and hierarchical protocols suffer a waste in terms of computing resources, whenever living processes have to rollback and recover from a checkpoint in order to tolerate failures. These protocols may also lead to I/O congestion when too many processes are checkpointing at the same time.

Message logging increases memory consumption, checkpointing time, and slows down failure-free execution when messages are logged. Our objective is to identify which protocol delivers the best performance for a given application on a given platform. While several criteria could be considered to make such a selection, we focus on the most widely used metric, namely, the expectation of the total parallel execution time.

Fault-tolerant protocols have different overheads in fault-free and recovery situations. These overheads depend on many factors (type of protocols, application characteristics, system features, etc.) that introduce complexity and limit the scope of experimental comparisons conducted in the past [16, 17]. In this chapter, we approach the fault tolerant protocol comparison from an analytical perspective. Our objective is to provide an accurate performance model covering the most suitable rollback recovery protocols for HPC. This model captures many optimizations proposed in the literature, but can also be used to explore the effects of novel optimizations, and to highlight the most critical parameters to be considered when evaluating a protocol.

The main contributions of this chapter are: (1) a comprehensive model that captures many rollback recovery protocols, including coordinated checkpoint, uncoordinated checkpoint, and the composite hierarchical hybrids; (2) a closed-form formula for the waste of computing resources incurred by each protocol. This formula is the key to assessing existing and new protocols, and constitutes the first tool that can help the community to compare protocols at very large scale, and to guide design decisions for given application/platform pairs; and (3) an instantiation of the model on several realistic scenarios involving state-of-the-art and future Exascale platforms, thereby providing practical insight and guidance.

This chapter is organized as follows. In Section 2.2, we describe our model that unifies coordinated rollback recovery approaches, and effectively captures coordinated, partially and totally uncoordinated approaches as well as many of their optimizations. We then use the model to analytically assess the performance of rollback recovery protocols. We instantiate the model with realistic scenarios in Section 2.3, and we present corresponding results in Section 2.4. These results are corroborated by a set of simulations (Section 2.5), demonstrating the accuracy of the proposed unified analytical model. Finally, we conclude and present perspectives in Section 2.6.

2.2 Model and Analytical Assessment

In this section, we discuss the unified model, together with the closed-form formulas for the waste optimization problem. We start with the description of the abstract model (Section 2.2.1). Processors are partitioned into G groups, where each group checkpoints independently and periodically. To help follow the technical derivation of the waste, we start with one group (Section 2.2.2) before tackling the general problem with $G \geq 1$ groups (Section 2.2.3), first under simplified assumptions, before tackling last the fully general model, which requires three additional parameters (payload overhead, faster execution replay after a failure, and increase in checkpoint size due to logging). We end up with a complicated formula that characterizes the waste of resources due to checkpointing. This formula can be instantiated to account for checkpointing protocols, see Section 2.3 for examples. Note that in all scenarios, we model the behavior of tightly coupled applications, meaning that no computation can progress on the entire platform as long as the recovery phase of a group with a failing processor is not completed.

2.2.1 Abstract model

In this section, we detail the main parameters of the model. We consider an application that executes on p_{total} processors.

Units– To avoid introducing several conversion parameters, we represent all the parameters of the model in seconds. The failure inter-arrival times, the duration of a downtime, checkpoint, or recovery are all expressed in seconds. Furthermore, we assume (without loss of generality) that one work unit is executed in one second, when all processors are computing at full rate. One work-unit may correspond to any relevant application-specific quantity. When a processor is slowed-down by another activity related to fault-tolerance (writing checkpoints to stable storage, logging messages, etc.), one work-unit takes longer than a second to complete.

Failures and MTBF– The platform consists of p_{total} identical processors. We use the term "processor" to indicate any individually scheduled compute resource (a core, a socket, a cluster node, etc) so that our work is agnostic to the granularity of the platform. These processors are subject to failures. Exponential failures are widely used for theoretical studies, while Weibull or log-normal failures are representative of the behavior of real-world platforms [64, 63, 65, 66]. The mean time between failures of a given processor is a random variable with mean ($MTBF$) μ (expressed in seconds). Given the failure distribution of one processor, it is difficult to compute, or even approximate, the failure distribution of a platform with p_{total} processors, because it is the *superposition* of p_{total} independent and identically distributed distributions (with a single processor). However, there is an easy formula for the MTBF of that distribution, namely $\mu = \frac{\mu}{p_{total}}$.

In our theoretical analysis, we do not assume to know the failure distribution of the platform, except for its mean value (the MTBF). Nevertheless, consider any time-interval $\mathcal{I} = [t, t+T]$ of length T and assume that a failure strikes during this interval. We can safely state that the probability for the failure to strike during any sub-interval $[t', t'+X] \subset \mathcal{I}$ of length X is $\frac{X}{T}$. Similarly, we state that the expectation of the time m at which the failure strikes is $m = t + \frac{T}{2}$. Neither of these statements rely on some specific property of the failure distribution, but instead are a direct consequence of averaging over all possible interval starting points, that will correspond to the beginning of checkpointing periods, and that are independent of failure dates.

Tightly-coupled application– We consider a tightly-coupled application executing on the p_{total} processors. Inter-processor messages are exchanged throughout the computation, which can only progress if all processors are available. When a failure strikes a processor, the application is missing one resource for a certain period of time of length D, the *downtime*. Then, the application recovers from the last checkpoint (*recovery* time of length R) before it re-executes the work done since that checkpoint and up to the failure. Under a hierarchical scenario, the useful work resumes only when the faulty group catches up with the overall state of the application at failure time. Many scientific applications are tightly-coupled and obey such a recovery scheme. Typically, the tightly-coupled application will be an iterative application with a global synchronization point at the end of each iteration. However, the fact that inter-processor information is exchanged continuously or at given synchronization steps (as in BSP-like models) is irrelevant: in steady-state mode, all processors must be available concurrently for the execution to actually progress. While the tightly-coupled assumption may seem very constraining, it captures the fact that processes in the application depend on each other and exchange messages at a rate exceeding the periodicity of checkpoints, preventing independent progress.

Blocking or non-blocking checkpoint– There are various scenarios to model the cost of checkpointing, so we use a flexible model, with several parameters to specify. The first question is whether checkpoints are blocking or not. On some architectures, we may have to stop executing the application before writing to the stable storage where the checkpoint data is saved; in that case checkpoint is fully blocking. On other architectures, checkpoint data can be saved on the fly into a local memory before the checkpoint is sent to the stable storage, while computation can resume progress; in that case, checkpoints can be fully overlapped with computations. To deal with all situations, we introduce a slow-down factor α: during a checkpoint of duration C, the work that is performed is αC work units, instead of C work-units if only computation takes place. In other words, $(1-\alpha)C$ work-units are wasted due to checkpoint jitter perturbing the progress of computation. Here, $0 \leq \alpha \leq 1$ is an arbitrary parameter. The case $\alpha = 0$ corresponds to a fully blocking checkpoint, while $\alpha = 1$ corresponds to a fully overlapped checkpoint, and all intermediate situations can be represented.

Periodic checkpointing strategies– For the sake of clarity and tractability, we focus on periodic scheduling strategies where checkpoints are taken at regular intervals, after some fixed amount of work-units have been performed. This corresponds to an infinite-length execution partitioned into periods of duration T. Without loss of generality, we partition T into $T = W + C$, where W is the amount of time where only computations take place, while C corresponds to the amount of time where checkpoints are taken.

Let $\text{TIME}_{\text{base}}$ be the application execution time without any fault tolerance mechanism and without failures. If we assume that TIME_{FF} is the execution time when checkpoints are introduced and WASTE_{FF} is the waste due to checkpoints, $\text{TIME}_{\text{base}}$ would be equal to TIME_{FF} minus the waste due to checkpoints, thus:

$$(1 - \text{WASTE}_{\text{FF}})\text{TIME}_{\text{FF}} = \text{TIME}_{\text{base}} \tag{2.1}$$

With the same idea, if we assume that $\text{TIME}_{\text{final}}$ is the time needed to complete the execution with failures and fault tolerance techniques:

$$(1 - \text{WASTE}_{\text{fail}})\text{TIME}_{\text{final}} = \text{TIME}_{\text{FF}} \tag{2.2}$$

By replacing the equation 2.2 in the equation 2.1 and if we assume that WASTE is the total waste:

$$(1 - \text{WASTE}_{\text{FF}})(1 - \text{WASTE}_{\text{fail}})\text{TIME}_{\text{final}} = \text{TIME}_{\text{base}} \tag{2.3}$$

We define WASTE as being the amount of time not performing useful computations,

$$\text{WASTE} = (\text{TIME}_{\text{final}} - \text{TIME}_{\text{base}})/\text{TIME}_{\text{final}} \tag{2.4}$$

Finally, we deduce the following formula for the global waste:

$$\text{WASTE} = 1 - (1 - \text{WASTE}_{\text{FF}})(1 - \text{WASTE}_{\text{fail}}) \tag{2.5}$$

If not slowed down for other reasons by the fault tolerant protocol (Section 2.2.3), the total amount of work units that are executed during a period of length T is thus $\text{WORK} = W + \alpha C$ (recall that there is a slow-down due to the overlap). In a failure-free environment, the *waste* of computing resources due to checkpointing is

$$\text{WASTE}_{\text{FF}} = \frac{T - \text{WORK}}{T} = \frac{(1-\alpha)C}{T} \tag{2.6}$$

As expected, if $\alpha = 1$ there is no overhead, but if $\alpha < 1$ (actual slowdown, or even blocking if $\alpha = 0$), checkpointing comes with a price in terms of performance degradation.

For the time being, we do not further quantify the length of a checkpoint, which is a function of several parameters. Instead, we proceed with the abstract model. We envision several scenarios in Section 2.3, only after setting up the formula for the waste in a general context.

Processor groups– As described above, we assume that the platform is partitioned into G groups of the same size. Each group contains q processors, hence $p_{total} = Gq$. When $G = 1$, we speak of a *coordinated* scenario, and we simply write C, D and R for the duration of a checkpoint, downtime and recovery. When $G \geq 1$, we speak of a *hierarchical* scenario. Each group of q processors checkpoints independently and sequentially in time $C(q)$. Similarly, we use $D(q)$ and $R(q)$ for the durations of the downtime and recovery. Of course, if we set $G = 1$ in the (more general) *hierarchical* scenario, we retrieve the value of the waste for the coordinated scenario. As already mentioned, we derive a general expression for the waste for both scenarios, before further specifying the values of $C(q)$, $D(q)$, and $R(q)$ as a function of q and the various architectural parameters under study.

2.2.2 Waste for the coordinated scenario ($G = 1$)

The goal of this section is to quantify the expected waste in the coordinated scenario where $G = 1$. The waste is the fraction of time that the processors do not compute at full rate, either because they are checkpointing, or because they are recovering from a failure. Recall that we write C, D, and R for the checkpoint, downtime, and recovery using a single group of p_{total} processors. We obtain the following equation for the waste, which we explain briefly below explanation is available to the reader and illustrate with Figure 2.1:

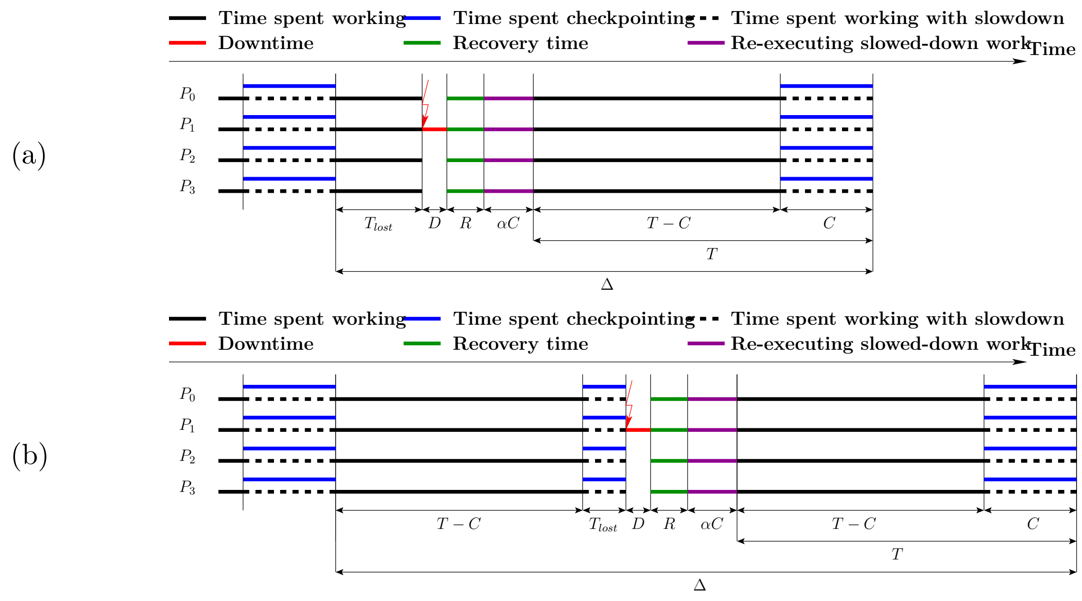


Figure 2.1: Coordinated checkpoint: illustrating the waste when a failure occurs (a) during the work phase; and (b) during the checkpoint phase.

$$
\begin{aligned}
\text{WASTE}_{\text{FF}} &= \frac{(1-\alpha)C}{T} && (2.7)\\
\text{WASTE}_{\text{fail}} &= \frac{1}{\mu}\Bigg(R + D + && (2.8)\\
&\quad \frac{T-C}{T}\left[\alpha C + \frac{T-C}{2}\right] && (2.9)\\
&+ \frac{C}{T}\left[\alpha C + T - C + \frac{C}{2}\right]\Bigg) && (2.10)
\end{aligned}
$$

• (2.7) is the portion of the execution lost in checkpointing, even during a fault-free execution, see Equation (2.6).
• (2.9) is the overhead of the execution time due to a failure during work interval $(T-C)$(see Figure 2.1(a)).
• (2.10) is the overhead due to a failure during a checkpoint (see Figure 2.1(b)).

After simplification of Equations (2.7) to (2.10), we get:

$$\text{WASTE}_{\text{fail}} = \frac{1}{\mu}\left(D + R + \frac{T}{2} + \alpha C\right) \tag{2.11}$$

Plugging this value back into Equation (2.5) leads to:

$$\text{WASTE}_{\text{coord}} = 1 - (1 - \frac{(1-\alpha)C}{T})(1 - \frac{1}{\mu}\left(D + R + \frac{T}{2} + \alpha C\right)) \tag{2.12}$$

We point out that Equation (2.12) is valid only when $T \ll \mu$: indeed, we made a first-order approximation when implicitly assuming that we do not have more than one failure during the same period. This hypothesis is required to allow the expression of the model in a closed form. In fact, the number of failures during a period of length T can be modeled as a Poisson process of parameter $\frac{T}{\mu}$; the probability of having $k \geq 0$ failures is $\frac{1}{k!}(\frac{T}{\mu})^k e^{-\frac{T}{\mu}}$. Hence the probability of having two or more failures is $\pi = 1 - (1 + \frac{T}{\mu})e^{-\frac{T}{\mu}}$. Enforcing the constraint $T \leq 0.1\mu$ leads to $\pi \leq 0.005$, hence a valid approximation when capping T to that value. Indeed, we have overlapping faults every 200 periods in average, so that our model is accurate for 99.5% of the checkpointing segments, hence it is quite reliable.

In addition to the previous constraint, we must enforce the condition $C \leq T$, by construction of the periodic checkpointing policy. Without the constraint $C \leq T \leq 0.1\mu$, the optimal checkpointing period T_{opt} that minimizes the expected waste in Equation (2.12) is $T_{\text{opt}} = \sqrt{2(1-\alpha)(\mu - (D+R))C}$. However, this expression for T_{opt} (which is known as Young's approximation [53] when $\alpha = 0$) may well be out of the admissible range. Finally, note that the optimal waste may never exceed 1, since it represents the fraction of time that is "wasted". In this latter case, the application no longer makes progress.

2.2.3 Waste for the hierarchical scenario ($G \geq 1$)

In this section, we compute the expected waste for the hierarchical scenario. We have G groups of q processors, and we let $C(q)$, $D(q)$, and $R(q)$ be the duration of the checkpoint, downtime, and recovery for each group. We assume that the checkpoints of the G groups take place in sequence within a period (see Figure 2.2(a)). We start by generalizing the formula obtained for the coordinated scenario before introducing several new parameters to the model.

Generalizing previous scenario with $G \geq 1$

We obtain the following intricate formula for the waste, which we illustrate with Figure 2.2 and the discussion below:

$$\text{WASTE}_{\text{hier}} = 1 - \left(1 - \frac{T - \text{WORK}}{T}\right)\left(1 - \frac{1}{\mu}\left(D(q) + R(q) + \text{RE-EXEC}\right)\right) \tag{2.13}$$

$$\text{WORK} = T - (1-\alpha)GC(q) \tag{2.14}$$

$$\text{RE-EXEC} =$$

$$\frac{T - GC(q)}{T}\frac{1}{G}\sum_{g=1}^{G}\left[(G-g+1)\alpha C(q) + \frac{T-GC(q)}{2}\right] \tag{2.15}$$

$$+ \frac{GC(q)}{T}\frac{1}{G^2}\sum_{g=1}^{G}\Bigg[\tag{2.16}$$

$$\sum_{s=0}^{g-2}(G-g+s+2)\alpha C(q) + T - GC(q) \tag{2.17}$$

$$+ G\alpha C(q) + T - GC(q) + \frac{C(q)}{2} \tag{2.18}$$

$$+ \sum_{s=1}^{G-g}(s+1)\alpha C(q)\Bigg] \tag{2.19}$$

• The first term in Equation (2.13) represents the overhead due to checkpointing during a fault-free execution (same reasoning as in Equation (2.6)), and the second term the overhead incurred in case of failure.
• (2.14) provides the amount of work units executed within a period of length T.
• (2.15) represents the time needed for re-executing the work when the failure happens in a work-only area, i.e., during the first $T - GC(q)$ seconds of the period (see Figure 2.2(a)).
• (2.16) deals with the case where the fault happens during a checkpoint, i.e. during the last $GC(q)$ seconds of the period (hence the first term that represents the probability of this event). We distinguish three cases, depending upon what group was checkpointing at the time of the failure:
- (2.17) is for the case when the fault happens before the checkpoint of group g (see Figure 2.2(b)).
- (2.18) is for the case when the fault happens during the checkpoint of group g (see Figure 2.2(c)).
- (2.19) is the case when the fault happens after the checkpoint of group g, during the checkpoint of group $g + s$, where $g + 1 \leq g + s \leq G$ (See Figure 2.2(d)).

Of course this expression reduces to Equation (2.12) when $G = 1$. Just as for the coordinated scenario, we enforce the constraint

$$GC(q) \leq T \leq 0.1\mu \tag{2.20}$$

The first condition is by construction of the periodic checkpointing policy, and the second is to enforce the validity of the first-order approximation (at most one failure per period).

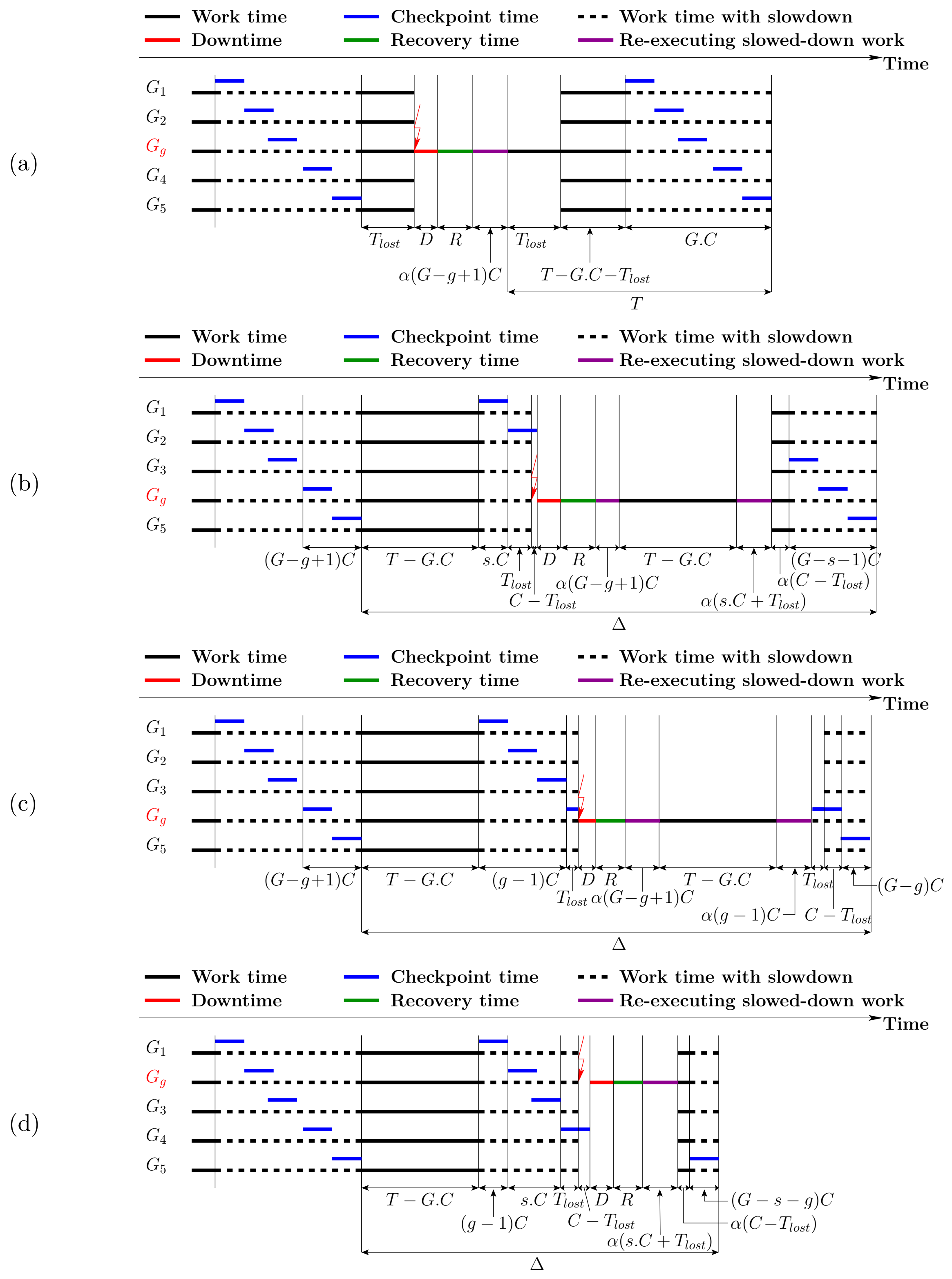


Figure 2.2: Hierarchical checkpoint: illustrating the waste when a failure occurs (a) during the work phase (Equation (2.15)); and during the checkpoint phase (Equations (2.16)–(2.19)), with three sub-cases: (b) before the checkpoint of the failing group (Equation (2.17)), (c) during the checkpoint of the failing group (Equation (2.18)), or (d) after the checkpoint of the failing group (Equation (2.19)).

Refining the model

We introduce three new parameters to refine the model when the processors have been partitioned into several groups. These parameters are related to the impact of message logging on execution, re-execution, and checkpoint image size, respectively.

Impact of message logging on execution and re-execution– With several groups, inter-group messages need to be stored in local memory as the execution progresses, and event logs must be stored in reliable storage, so that the recovery of a given group, after a failure, can be done independently of the other groups. This induces an overhead, which we express as a slowdown of the execution rate: instead of executing one work-unit per second, the application executes only λ work-units, where $0 < \lambda < 1$. Typical values for λ are said to be $\lambda \approx 0.98$, meaning that the overhead due to payload messages is only a small percentage [22, 23].

On the contrary, message logging has a positive effect on re-execution after a failure, because inter-group messages are stored in memory and directly accessible after the recovery. Our model accounts for this by introducing a speed-up factor ρ during the re-execution. Typical values for ρ lie in the interval $[1; 2]$, meaning that re-execution time can be reduced by up to half for some applications [16].

Fortunately, the introduction of λ and ρ is not difficult to account for in the expression of the expected waste: in Equation (2.13), we replace $\textsc{Work}$ by $\lambda\textsc{Work}$ and $\textsc{Re-Exec}$ by $\frac{\textsc{Re-Exec}}{\rho}$ and obtain

$$\textsc{Waste}_{\text{hier}} = 1 - \left(1 - \frac{T - \lambda\textsc{Work}}{T}\right)\left(1 - \frac{1}{\mu}\left(D(q) + R(q) + \frac{\textsc{Re-Exec}}{\rho}\right)\right) \tag{2.21}$$

where the values of $\textsc{Work}$ and $\textsc{Re-Exec}$ are unchanged, and given by Equations (2.14) and (2.15 – 2.19) respectively.

Impact of message logging on checkpoint size– Message logging has an impact on the execution and re-execution rates, but also on the size of the checkpoint. Because inter-group messages are logged, the size of the checkpoint increases with the amount of work per unit. Consider the hierarchical scenario with G groups of q processors. Without message logging, the checkpoint time of each group is $C_0(q)$, and to account for the increase in checkpoint size due to message logging, we write the equation

$$C(q) = C_0(q)(1 + \beta\lambda\textsc{Work}) \Leftrightarrow \beta = \frac{C(q) - C_0(q)}{C_0(q)\lambda\textsc{Work}} \tag{2.22}$$

As before, $\lambda\textsc{Work} = \lambda(T - (1 - \alpha)GC(q))$ (see Equation (2.14)) is the number of work units, or application iterations, completed during the period of duration T, and the parameter β quantifies the increase in the checkpoint image size per work unit, as a proportion of the application footprint. Typical values of β are given in the examples of Section 2.3. Combining with Equation (2.22), we derive the value of $C(q)$ as

$$C(q) = \frac{C_0(q)(1 + \beta\lambda T)}{1 + GC_0(q)\beta\lambda(1 - \alpha)} \tag{2.23}$$

The first constraint in Equation (2.20), namely $GC(q) \leq T$, now translates into $\frac{GC_0(q)(1+\beta\lambda T)}{1+GC_0(q)\beta\lambda(1-\alpha)} \leq T$, hence

$$GC_0(q)\beta\lambda\alpha \leq 1 \text{ and } T \geq \frac{GC_0(q)}{1 - GC_0(q)\beta\lambda\alpha} \tag{2.24}$$

2.3 Case Studies

In this section, we use the previous model to evaluate different case studies. We propose three generic scenarios for the checkpoint protocols, three application examples with different values for the parameter β, and four platform instances.

2.3.1 Checkpointing algorithm scenarios

COORD-IO – The first scenario considers a coordinated approach, where the duration of a checkpoint is the time needed for the p_{total} processors to write the memory footprint of the application onto stable storage. Let Mem denote this memory, and b_{io} represents the available I/O bandwidth. Then

$$C = C_{\mathrm{Mem}} = \frac{\mathrm{Mem}}{\mathrm{b}_{io}} \tag{2.25}$$

In most cases we have equal write and read speed access to stable storage, and we let $R = C = C_{\mathrm{Mem}}$, but in some cases we have different values, for example with the K-Computer (see Table 2.1). As for the downtime, the value D is the expectation of the duration of the downtime. With a single processor, the downtime has a constant value, but with several processors, the duration of the downtime is difficult to compute: a processor can fail while another one is down, thereby leading to cascading downtimes. The exact value of the downtime with several processors is unknown, even for failures distributed according to an exponential law; but in most practical cases, the lower bound given by the downtime of a single processor is expected to be very accurate, and we use a constant value for D in our case studies.

HIERARCH-IO – The second scenario uses a number of relatively large groups. Typically, these groups are composed to take advantage of the application communication pattern [22, 21]. For instance, if the application executes on a 2D-grid of processors, a natural way to create processor groups is to have one group per row (or column) of the grid. If all processors of a given row belong to the same group, horizontal communications are intra-group communications and need not to be logged. Only vertical communications are inter-group communications and need to be logged.

With large groups, there are enough processors within each group to saturate the available I/O bandwidth, and the G groups checkpoint sequentially. Hence the total checkpoint time without message logging, namely $GC_0(q)$, is equal to that of the coordinated approach. This leads to the simple equation

$$C_0(q) = \frac{C_{\mathrm{Mem}}}{G} = \frac{\mathrm{Mem}}{G\mathrm{b}_{io}} \tag{2.26}$$

where Mem denotes the memory footprint of the application, and b_{io} the available I/O bandwidth. Similarly as before, we use $R(q)$ for the recovery (either equal to $C(q)$ or not), and a constant value $D(q) = D$ for the downtime.

HIERARCH-PORT – The third scenario investigates the possibility of having a large number of very small groups, a strategy proposed to take advantage of hardware proximity and failure probability correlations [20]. However, if groups are reduced to a single processor, a single checkpointing group is not sufficient to saturate the available I/O bandwidth. In this strategy, multiple groups of q processors are allowed to checkpoint simultaneously in order to saturate the I/O bandwidth. We define $\mathrm{q}_{\mathrm{min}}$ as the smallest value such that $\mathrm{q}_{\mathrm{min}}\mathrm{b}_{port} \geq \mathrm{b}_{io}$, where b_{port} is the network bandwidth of a single processor. In other words, $\mathrm{q}_{\mathrm{min}}$ is the minimal size of groups so that Equation (2.26) holds.

Small groups typically imply logging more messages (hence a larger growth factor of the checkpoint per work unit β, and possibly a larger impact on computation slowdown λ). For an application executing on a 2D-grid of processors, twice as many communications will be logged (assuming a symmetrical communication pattern along each grid direction). However, let us compare recovery times in the HIERARCH-PORT and HIERARCH-IO strategies; assume that $R_0(q) = C_0(q)$ for simplicity. In both cases Equation (2.26) holds, but the number of groups is significantly larger for HIERARCH-PORT, thereby ensuring a much shorter recovery time.

2.3.2 Application examples

We study the increase in checkpoint size due to message logging by detailing three application examples that are typical scientific applications executing on 2D-or 3D-processor grids, but that exhibit a different checkpoint increase rate parameter β.

2D-STENCIL– We first consider a 2D-stencil computation: a real matrix of size $n \times n$ is partitioned across a $p \times p$ processor grid, where $p^2 = p_{total}$. At each iteration, each element is averaged with its 8 closest neighbors, requiring rows and columns that lie at the boundary of the partition to be exchanged (it is easy to generalize to larger update masks). Each processor holds a matrix block of size $b = n/p$, and sends four messages of size b (one in each grid direction) . Then each element is updated, at the cost of 9 double floating-point operations. The (parallel) work for one iteration is thus WORK $= \frac{9b^2}{\mathrm{s}_p}$, where s_p is the speed of one processor.

Here Mem $= 8n^2$ (in bytes), since there is a single (double real) matrix to store. As already mentioned, a natural (application-aware) group partition is with one group per row (or column) of the grid, which leads to $G = q = p$. Such large groups correspond to the HIERARCH-IO scenario, with $C_0(q) = \frac{C_{\mathrm{Mem}}}{G}$. At each iteration, vertical (inter-group) communications are logged, but horizontal (intra-group) communications are not logged. The size of logged messages is thus $2pb = 2n$ for each group. If we checkpoint after each iteration, $C(q) - C_0(q) = \frac{2n}{\mathrm{b}_{io}}$, and we derive from Equation (2.22) that $\beta = \frac{2np\mathrm{s}_p}{n^2 9b^2} = \frac{2\mathrm{s}_p}{9b^3}$. We stress that the value of β is unchanged if groups checkpoint every k iterations, because both $C(q) - C_0(q)$ and WORK are multiplied by a factor k. Finally, if we use small groups of size $\mathrm{q}_{\mathrm{min}}$, we have the HIERARCH-PORT scenario. We still have $C_0(q) = \frac{C_{\mathrm{Mem}}}{G}$, but now the value of β has doubled since we log twice as many communications.

MATRIX-PRODUCT– Consider now a typical linear-algebra kernel involving matrix products. For each matrix-product, there are three matrices involved, so Mem $= 24n^2$ (in bytes). The matrix partition is similar to previous scenario, but now each processor holds three matrix blocks of size $b = n/p$. Consider Cannon's algorithm [13] which has p steps to compute a product. At each step, each processor shifts one block vertically and one block horizontally, and WORK $= \frac{2b^3}{\mathrm{s}_p}$. In the HIERARCH-IO scenario with one group per grid row, only vertical messages are logged: $\beta = \frac{\mathrm{s}_p}{6b^3}$. Again, β is unchanged if groups checkpoint every k steps, or every matrix product ($k = p$). In the COORD-PORT scenario with groups of size $\mathrm{q}_{\mathrm{min}}$, the value of β is doubled.

3D-STENCIL– This application is similar to 2D-STENCIL, but with a 3D matrix of size n partitioned across a 3D-grid of size p, where $8n^3 =$ Mem and $p^3 = p_{total}$. Each processor holds a cube of size $b = n/p$. At each iteration, each pixel is averaged with its 26 closest neighbors, and WORK $= \frac{27b^3}{\mathrm{s}_p}$. Each processor sends the six faces of its cube, one in each direction. In addition to COORD-IO, there are now three hierarchical scenarios: A) HIERARCH-IO-PLANE where groups are horizontal planes, of size p^2. Only vertical communications are logged, which represents two faces per processor: $\beta = \frac{2\mathrm{s}_p}{27b^3}$; B) Hierarch-IO-Line where groups are lines, of size p. Twice as many communications are logged, which represents four faces per processor: $\beta = \frac{4\mathrm{s}_p}{27b^3}$; C) Hierarch-Port (groups of size $\mathrm{q}_{\min}$). All communications are logged, which represents six faces per processor: $\beta = \frac{6\mathrm{s}_p}{27b^3}$. The order of magnitude of b is the cubic root of the memory per processor for 3D-Stencil, while it was its square root for 2D-Stencil and Matrix-Product, so β will be larger for 3D-Stencil.

2.3.3 Platforms and parameters

We consider multiple platforms, existing or envisioned, that represent state-of-the-art targets for HPC applications. Table 2.1 presents the basic characteristics of the platforms we consider. The machine named Titan represents the fifth phase of the Jaguar supercomputer, as presented by the Oak Ridge Leadership Computing Facility (http://www.olcf.ornl.gov/computing-resources/titan/). The cumulated bandwidth of the I/O network is targeted to top out at 1 MB/s/core, resulting in 300GB/s for the whole system. We consider that all existing machines are limited for a single node output by the bus capacity, at approximately 20GB/s. The K-Computer machine, hosted by Riken in Japan, is the second fastest supercomputer of the Top 500 list at the time of writing. Its I/O characteristics are those presented during the Lustre File System User's Group meeting, in April, 2011 [12], with the same bus limitation for a single node maximal bandwidth. The two Exascale machines represent the two most likely scenarios envisioned by the International Exascale Software Project community [92], the largest variation being on the number of cores a single node should host. For all platforms, we let the speed of one core be 1 Gigaflops, and we derive the speed of one processor s_p by multiplying by the number of cores.

Table 2.1 also presents key parameters for all platform/scenario combinations. In all instances, we use the default values: $\rho = 1.5$, $\lambda = 0.98$ and $\alpha = 0.3$. These values lead to results representative of the trends observed throughout the set of tested values.

2.3.4 Checkpoint duration

The last parameter that we consider is the duration of the checkpoint. Equation (2.25) states that $C = C_{\text{Mem}} = \frac{\text{Mem}}{\mathrm{b}_{io}}$, hence the duration of the checkpoint is proportional to the volume of data written to stable storage, and inversely proportional to the cumulated I/O bandwidth that is available on the platform. As for the volume of data written, it can range from the entire memory available on the platform (for those applications whose footprint is maximal), down to a small percentage of this value. As for the cumulated I/O bandwidth, it can range from the values given in Table 2.1 down to a small fraction of these values, if one can use advanced checkpointing techniques, like incremental checkpointing [8, 9] to reduce the checkpoint size, or multi-level checkpointing [6], diskless checkpointing [10, 11], or new generation hardware, providing local remanent memory [7] to increase the I/O bandwidth. To account for the wide range of both parameters (volume and bandwidth), we propose several scenarios:

$C_{\max}$ In this scenario, which represents the worst case, the duration of a checkpoint is $C_{\max} = C = C_{\text{Mem}}$ as in Equation (2.25): here we use the values of Table 2.1, both for Mem (the application uses the entire platform memory, and no technique such as incremental checkpointing, or compressive checkpointing, can be used to reduce the amount of memory that needs to be saved), and for b_{io} (the checkpoint is stored in the remote reliable file storage system, and its transfer speed is limited by the I/O bandwidth of the platform);

$\frac{C_{\max}}{X}$ In these scenarios, the duration of a checkpoint is $\frac{C_{\max}}{X}$, where $X \in \{10, 100, 1000\}$. Note that this does not mean that a single technique allows to reduce the data volume by a factor X; instead, the ratio $\frac{\text{Mem}}{\text{b}_{io}}$ is divided by X, by combining all available techniques and hardware, and both the numerator (smaller volume) and the denominator (faster transfer) can contribute to the reduction of checkpoint duration. The objective is to investigate whether, and to what extent, faster checkpointing can prove useful, or necessary, at very large scale.

2.4 Results from the model

This section covers the results of our unified model on the previously described scenarios. In order to grant fellow researchers access to the model, results and scenarios proposed in this chapter, we made our computation spreadsheet available at http://icl.cs.utk.edu/~herault/UnifiedModel/.

We start with some words of caution. First, the applications used for this evaluation exhibit properties that makes them a difficult case for hierarchical checkpoint/restart techniques. These applications are communication intensive, which leads to a noticeable impact on performance (due to message logging). In addition, their communication patterns create logical barriers that make them tightly-coupled, giving a relative advantage to all coordinated checkpointing methods (due to the lack of independent progress). However, these applications are more representative of typical HPC applications than loosely-coupled (or even independent) jobs, and their communication-to-computation ratio tends to zero with the problem size (full weak scalability). Next, we point out that the theoretical values used in the model instances, and summarized in Table 2.1, are overly optimistic, based on the values released by the constructors and not on measured values. Finally, we stress that the horizontal axis of all figures is the processor MTBF μ, which ranges from 1 year to 100 years, a choice consistent with the usual representation in the community.

Section 2.3 above presented in detail how the values of Table 2.1 were obtained. We started with the basic numbers of the different platforms (number of cores, processors, amount of memory and I/O capacity), and derived for each platform and Scenario the corresponding number of groups (for hierarchical protcols), their size, and from this the costs of taking a group checkpoint. We then derived, for each target application, the value of β, which depends on the group size.

The first observation is that when $C = C_{\max}$, only Titan is a useful platform! Indeed, we obtain a waste equal to 1 for all scenario/application combinations, throughout the whole range of the MTBF μ, for both the K-Computer or Exascale platforms. This was expected and simply shows that for such large platforms, the checkpoint time must be significantly smaller than $C_{\max}$, the time needed to write the entire platform memory onto stable storage.

Along the same line, we only report values for $C = C_{\max}$ on Titan, for $C = \frac{C_{\max}}{10}$ on Titan and the K-computer, for $C = \frac{C_{\max}}{100}$ on Exascale-Fat, and for $C = \frac{C_{\max}}{1{,}000}$ on Exascale-Fat and Exascale-Slim. Unreported values correspond to situations where the checkpoint duration is too large for the platform to be useful. A few comments apply to all platforms:

— Hierarchical protocols are very sensitive to message logging: a direct relationship between β, and the observed waste can be seen when moving from one application to another, and even for different protocols within the same application.

— Hierarchical protocols tend to provide better results for small MTBFs. Thus, they seem more suitable for failure-prone platforms. While they struggle when the communication intensity increases (the case of the 3D-Stencil), they provide limited waste for all the other cases.

— The faster the checkpointing time, the smaller the waste. This conclusion is quite expected, but our results allow quantifying the gain.

On Titan, when using $C_{\max}$, the key factors impacting the balance between coordinated and hierarchical protocols are the communication intensity of the applications (2D-Stencil, Matrix-Product, and 3D-Stencil), and the I/O capabilities of the system. The coordinated protocol has a slow startup, preventing the application from progressing when the platform MTBF μ is under a system limit directly proportional to the time required to save the coordinated checkpoint. Hierarchical protocols have a faster startup. However, as the MTBF increases, the optimal interval between checkpoints increase, and the cost of logging the messages (and the increase in checkpoint size it implies) becomes detrimental to the hierarchical protocols (even considering the most promising approaches). The vertical segments on the graphs correspond to cut-off values where we enforce the condition $T \leq 0.1\mu$ (see Equation (2.20)). Values of μ for which no waste is reported correspond to configurations for which no period can satisfy Equation (2.20).

Moving to $\frac{C_{\max}}{10}$, the same remarks can be made about the shape of the figures. Compared to a checkpointing time of $C_{\max}$, the waste is significantly smaller, leading to a very good yield of the platform as soon as the MTBF μ exceeds 10 years. The K-Computer shows similar behavior. However, the waste is still important even for large MTBF values for all application scenarios. This can be attributed to the low I/O bandwidth and high amount of total memory of the parameters used for the K-computer, when compared to the parameters considered for the Titan setup.

Moving to Exascale platforms, the Exascale-Fat platform starts to show application progress when $\frac{C_{\max}}{100}$ is used. However, just like the K-Computer, the waste is still important even for large MTBF. When checkpointing becomes ten times faster, the results are more promising. The Exascale-Slim platform starts to be useful when using $\frac{C_{\max}}{1{,}000}$, which corresponds to checkpointing the application within a few seconds. Overall, Exascale-Fat leads to a smaller waste (or better resource usage) than Exascale-Slim; the main reason is that the Fat version has fewer processors, hence a larger platform MTBF. Indeed, the individual processor MTBF is assumed to be the same for both Exascale-Fat and Exascale-Slim, which may be unfair since there are 10 times more cores per node in the Fat version.

Setting aside the expected conclusion that an efficient process checkpointing strategy will be critical to enable rollback/recovery at exascale, the model leads to an important prediction for Exascale machines (Fat or Slim scenarios): unless extremely high reliability of the components can be guaranteed (MTBF per component of 30 to 50 years for the Stencil applications), hierarchical checkpointing approaches will (i) exhibit a lower waste than coordinated checkpointing, and (ii) allow for an efficiency two to four times higher than replication. This conclusion holds even for applications as tightly-coupled and as communication-intensive as the ones evaluated in this study.

2.5 Validation of the model

To validate the mathematical model, we wrote a simulator that generates a random trace of errors (parameterized by an Exponential failure distribution or a Weibull one with a shape parameter of 0.7 or 0.5). On this error trace, we simulate the behavior of the various fault tolerance protocols. In the simulator, there is no assumption on when errors can happen: an error can strike a processor while another or the same processor is already subject to a failure and during a recovery phase.

We arbitrarily set the failure-free duration of the parallel application execution to 4 days. This guarantees that enough failures happen during each simulation run to evaluate the waste. We measure the simulated execution time of each application on each platform and on each error trace using a time-out of one year: if an execution does not complete before the one year deadline, we consider it never completes and do not report any result for this particular platform/application setting. From the simulated execution times, we compute the average waste.

All protocols use the same parameters in the simulator as the ones fed to the mathematical model: checkpoint durations, overheads of message logging and consequences in the checkpoint size, amount of work that can be done in parallel, are simulated by increasing accordingly the duration of the execution. The checkpoint interval is set in each case to the optimal value, as provided by the mathematical model. In order to evaluate the accuracy of the optimal checkpoint interval forecasted by the model, we also ran a set of experiments that investigate other random checkpoint interval values around the forecasted best value, and keep the best value in the experiments denoted BestPer.

Figure 2.4 reports the waste for various application/platform scenarios for a Weibull failure distribution with $k = 0.7$. Results for an Exponential failure distribution and for a Weibull with $k = 0.5$, are provided at `http://icl.cs.utk.edu/~herault/UnifiedModel/`. Each point on the graphs is an average over 20 randomly generated instances.

Overall, Figures 2.3 and 2.4 present similar trends and conclusions. The main differences are seen for low MTBFs, in the vicinity of cut-offs values for Figure 2.3. There, either the first-order assumption could no longer be satisfied, or coordinated protocols were assessed not to allow for any application progress. Simulations show that, in these extreme settings, our analytical study was pessimistic: coordinated protocols have indeed very bad performance, but often applications still make progress (albeit at unsatisfactory rate); coordinated protocols have an advantage over hierarchical protocols for slightly lower MTBFs than predicted. However, the simulations validate the relative performance, and the general efficiency, of the different protocols.

For each scenario and each protocol, we plot (in solid line) the average waste for the checkpointing period computed with our model (the one minimizing Equation (2.21)). In Figure 2.4 we also plot (in dotted line) the average waste obtained for the best checkpointing period (BestPer), numerically found by generating, and evaluating through simulations, a set of 480 periods representative of a very large neighborhood of the period computed with our model. The very good adequation between solid and dotted lines show that our model enables to compute near-optimal checkpointing periods, even when its underlying assumptions cannot be guaranteed.

2.6 Conclusion

Despite the increasing importance of fault tolerance in achieving sustained, predictable performance, the lack of models and predictive tools has restricted the analysis of fault tolerant protocols to experimental comparisons only, which are painfully difficult to realize in a consistent and repeated manner. This chapter introduces a comprehensive model of rollback recovery protocols that encompasses a wide range of checkpoint/restart protocols, including coordinated checkpoint and an assortment of uncoordinated checkpoint protocols (based on message logging). This model provides the first tool for a *quantitative* assessment of all these protocols.

Instantiation on future platforms enables the investigation and understanding of the behavior of fault tolerant protocols at scales currently inaccessible. The results presented in Section 2.4, and corroborated by Section 2.5, highlight the following tendencies:
• Hardware properties will have tremendous impact on the efficiency of future platforms. Under the early assumptions of the projected Exascale systems, rollback recovery protocols are mostly ineffective. In particular, significant efforts are required in terms of I/O bandwidth to enable any type of rollback recovery to be competitive. With the appropriate provision in I/O (or the presence of distributed storage on nodes), rollback recovery can be competitive and significantly outperform full-scale replication [69] (which by definition cannot reach more than 50% efficiency).
• Under the assumption that I/O network provision is sufficient, the reliability of individual processors has a minor impact on rollback recovery efficiency. This suggests that most research efforts, funding and hardware provisions should be directed to I/O performance rather than improving component reliability in order to increase the scientific throughout of Exascale platforms.
• The model outlines some realistic ranges where hierarchical checkpointing outperforms coordinated checkpointing, thanks to its faster recovery from individual failures. This early result had already been outlined experimentally at smaller scales, but it was difficult to project at future scales. Our study provides a theoretical foundation and a quantitative evaluation of the drawbacks of checkpoint/restart protocols at Exascale; it can be used as a first building block to drive the research field forward, and to design platforms with specific resilience requirements. Throughout the simulations, we have checked (by an extensive brute-force comparison) that our model could predict near-optimal checkpointing periods for the whole range of the protocol/platform/application combinations; this gives us very good confidence that this model will prove reliable and accurate in other frameworks. As we are far from a comprehensive understanding of future Exascale applications and platform characteristics, we hope that the community will be interested in instantiating our publicly available model with different scenarios and case-studies.

Name	Number of cores	Number of processors p_{total}	Number of cores per processor	Memory per processor
Titan	299,008	18,688	16	32GB
K-Computer	705,024	88,128	8	16GB
Exascale-Slim	1,000,000,000	1,000,000	1,000	64GB
Exascale-Fat	1,000,000,000	100,000	10,000	640GB

Name	I/O Network Bandwidth (b_{io})		I/O Bandwidth (b_{port})
	Read	Write	Read/Write per processor
Titan	300GB/s	300GB/s	20GB/s
K-Computer	150GB/s	96GB/s	20GB/s
Exascale-Slim	1TB/s	1TB/s	200GB/s
Exascale-Fat	1TB/s	1TB/s	400GB/s

Name	Scenario	G ($C(q)$)	β for 2D-Stencil	β for Matrix-Product
	Coord-IO	1 (2,048s)	/	/
Titan	Hierarch-IO	136 (15s)	0.0001098	0.0004280
	Hierarch-Port	1,246 (1.6s)	0.0002196	0.0008561
	Coord-IO	1 (14,688s)	/	/
K-Computer	Hierarch-IO	296 (50s)	0.0002858	0.001113
	Hierarch-Port	17,626 (0.83s)	0.0005716	0.002227
	Coord-IO	1 (68,719s)	/	/
Exascale-Slim	Hierarch-IO	1,000 (68.7s)	0.0002599	0.001013
	Hierarch-Port	200,000 (0.32s)	0.0005199	0.002026
	Coord-IO	1 (68,719s)	/	/
Exascale-Fat	Hierarch-IO	316 (217s)	0.00008220	0.0003203
	Hierarch-Port	33,333 (1.92s)	0.00016440	0.0006407

Name	Scenario	G	β for 3D-Stencil
	Coord-IO	1	/
Titan	Hierarch-IO-Plane	26	0.001476
	Hierarch-IO-Line	676	0.002952
	Hierarch-Port	1,246	0.004428
	Coord-IO	1	/
K-Computer	Hierarch-IO-Plane	44	0.003422
	Hierarch-IO-Line	1,936	0.006844
	Hierarch-Port	17,626	0.010266
	Coord-IO	1	/
Exascale-Slim	Hierarch-IO-Plane	100	0.003952
	Hierarch-IO-Line	10,000	0.007904
	Hierarch-Port	200,000	0.011856
	Coord-IO	1	/
Exascale-Fat	Hierarch-IO-Plane	46	0.001834
	Hierarch-IO-Line	2,116	0.003668
	Hierarch-Port	33,333	0.005502

Table 2.1: Basic characteristics of platforms used to instantiate the model, and all parameters for each platform/scenario combination. The equations $C_0(q) = C/G$ and $R_0(q) = R/G$ always hold.

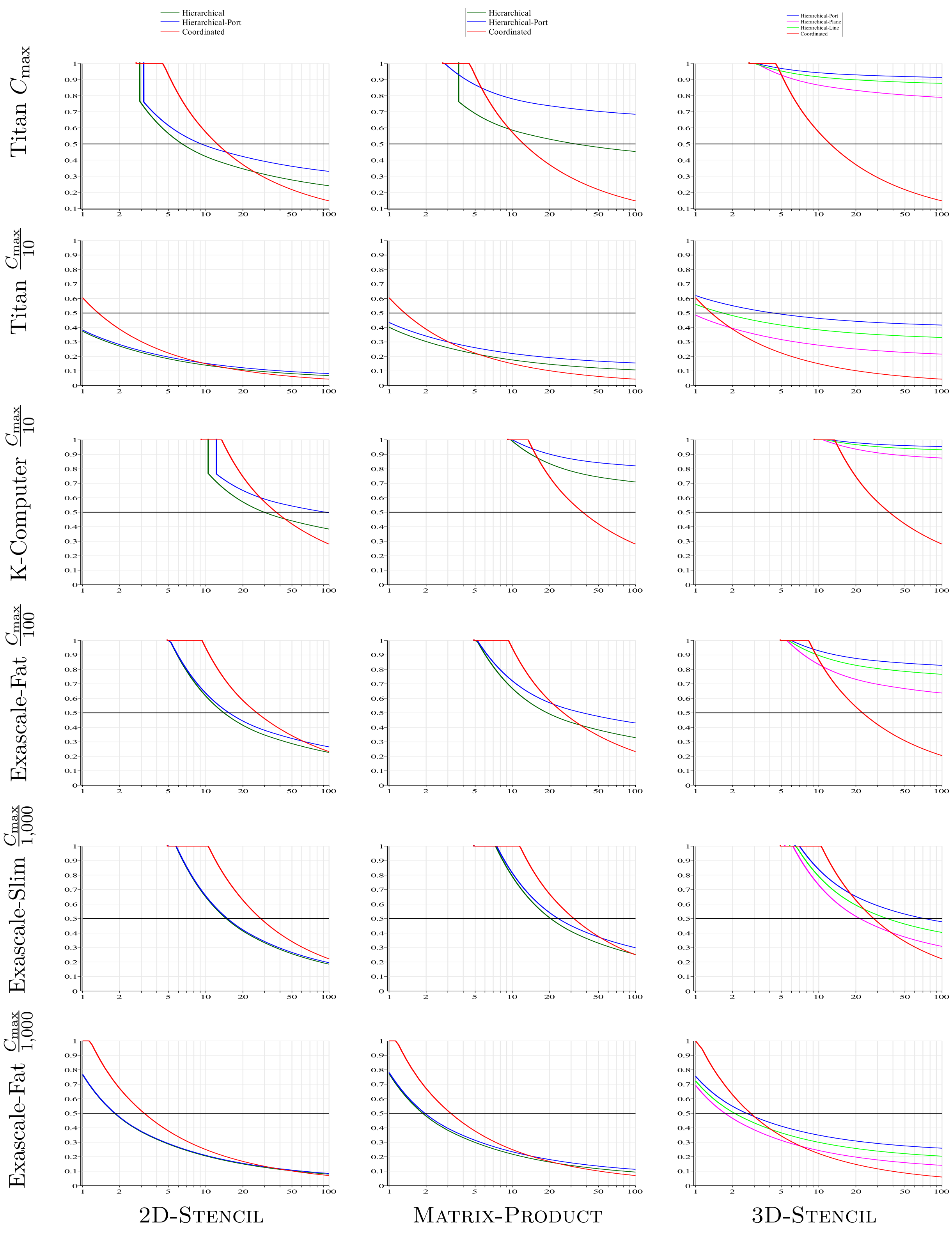


Figure 2.3: Model: waste as a function MTBF μ (years per processor).

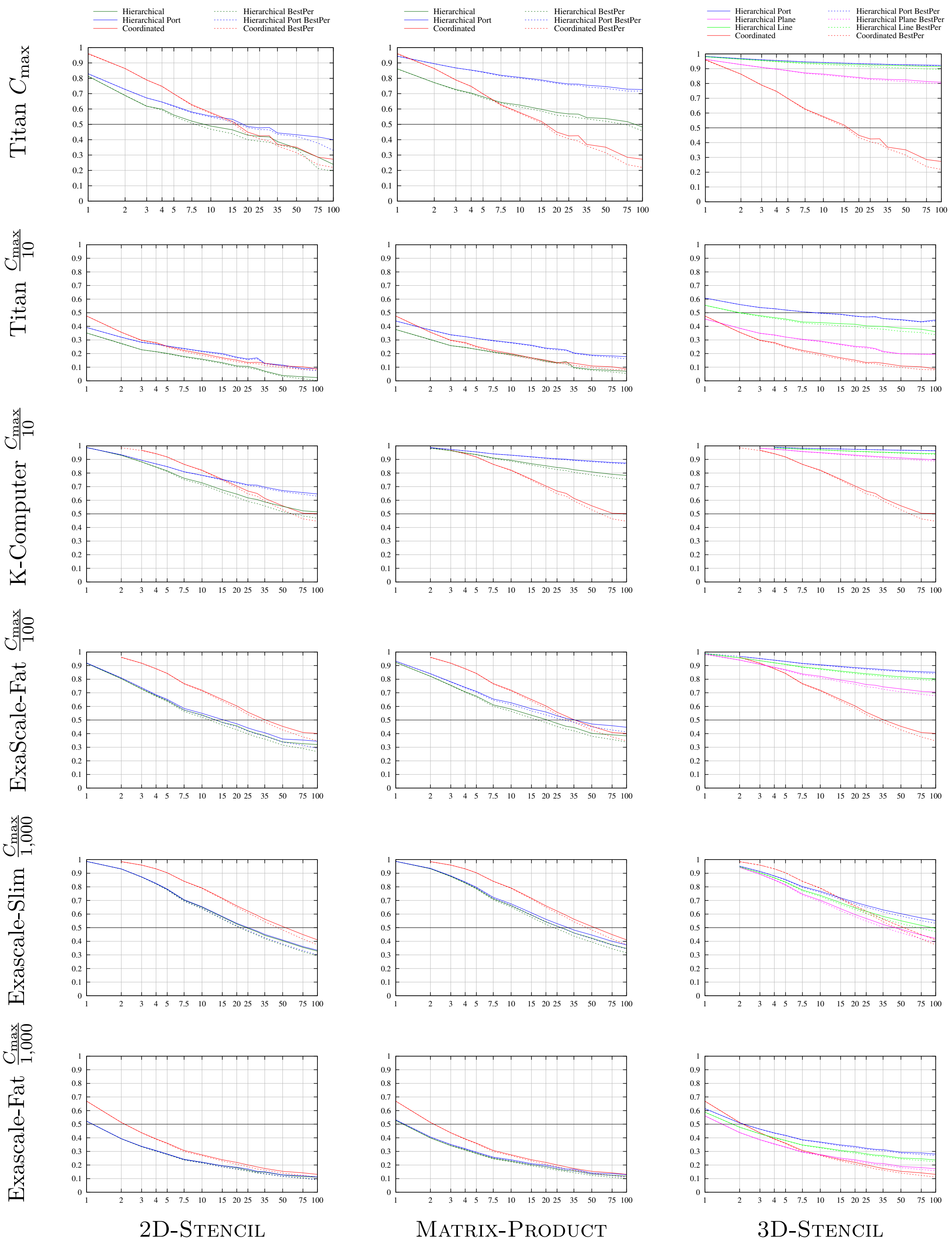


Figure 2.4: Waste as a function of processor MTBF μ, for a Weibull distribution with k=0.7

Chapter 3

Combining Replication and Coordinated Checkpointing

3.1 Introduction

For applications that enroll large numbers of processors, processor failures are projected to be common occurrences [24, 25, 26]. Failures occur because not all faults are automatically detected and corrected in current production hardware. To tolerate failures the standard approach is to use rollback and recovery for resuming application execution from a previously saved fault-free execution state, or *checkpoint*. In spite of these efforts, the necessary checkpoint frequency for tolerating failures in large-scale platforms can become so large that processors spend more time checkpointing than computing. This is because the MTBF (Mean Time Between Failures) of the platform becomes so small that the application performs too many recoveries and re-executions to make progress efficiently.

One possible solution to this problem is to increase the reliability of individual components, e.g., with more hardware redundancy. But this increase comes at a higher cost. Since system acquisition costs are typically constrained when designing a parallel platform, vendors must instead use commercial-of-the-shelf (COTS) components. The reliability of these COTS components is defined by the product lifetime, as driven by the market. HPC systems with COTS components will thus experience higher failure rates at higher scales [29], thereby limiting parallel efficiency if only checkpoint-recovery is used at these scales. Furthermore, even if the MTBF of an individual component is a high μ_{comp}, then the MTBF of a platform with p components is $\mu = \frac{\mu_{comp}}{p}$. No matter how reliable the individual components, there is thus a value of p above which errors are so frequent that they can prevent any application progress.

In this chapter we focus on *replication*: several processors perform the same computation synchronously, so that a fault on one of these processors does not lead to an application failure. Replication is an age-old fault-tolerant technique, but it has gained traction in the HPC context only relatively recently. While replication wastes compute resources in fault-free executions, it can alleviate the poor scalability of checkpoint-recovery.

We study two replication approaches. Consider a parallel application that is *moldable*, meaning that it can be executed on an arbitrary number of processors, which each processor running one application process. In the first approach, *group replication*, multiple application instances are executed. For example, 2 distinct n-process application instances could be executed on a $2n$-processor platform. Each instance runs at a smaller scale, meaning that it has better parallel efficiency than a single $2n$-process instance due to a smaller checkpointing frequency.

Furthermore, once an instance saves a checkpoint, another 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.

In the second approach, *process replication*, a single instance of an application is executed but each application process is (transparently) replicated. For the same example, one executes the application with n processes so that there are two replicas of each process, each running on a distinct physical processor. This approach is sensible because the mean time to failure of a group of two replicas is larger than that of a single processor, meaning that the checkpointing frequency can be lowered thus improving parallel efficiency. In [69] Ferreira et al. have studied process replication, with a practical implementation and some analytical results.

Process replication largely outperform group replication, simply because process replication leads to dramatically increased MTBF for each replica set. However, process replication may not always be a feasible option. This is because process replication must be provided transparently as part of the runtime system. There are several popular programming models and runtimes (e.g., message passing, concurrent objects, distributed components, workflows, algorithmic skeletons). In some cases, e.g., for the Message Passing Interface (MPI) runtime, proof-of-concept implementations that provide process replication are available [69]. But in general, many existing and popular runtimes do not (yet) provide transparent process replication for the purpose of fault-tolerance, and enhancing them with this capability may be non trivial. A solution could be to implement process replication explicitly as part of the application, but this would be labor-intensive, especially for legacy applications. Group replication can be used whenever process replication is not available because it 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 moldable and that an instance be startable from a saved checkpoint file.

We note that (process or group) replication prevents the execution of an application that requires the aggregate memory of the full platform, and in this sense limits the scale of the application execution. However, such full-scale execution is likely impractical in the first place due to the need for a high checkpointing frequency. The processors would spend more time saving state than computing state, thus leading to low parallel efficiency.

At first glance, it may seem paradoxical that better performance can be achieved by using (process or group) replication. After all in the above example, 50% of the platform is "wasted" to perform redundant computation. As a result the application instance runs at a smaller scale. But, precisely because the scale is smaller, the application can use a lower checkpointing frequency, and can thus have better parallel efficiency when compared to an application instance running at full scale. 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 the reduced checkpointing frequency.

In this chapter we study group and process replication from a theoretical perspective, with the following highlights:

— For group replication:
  — We propose a simple yet effective algorithm for group replication.
  — For exponentially distributed failures, we derive a checkpointing period that minimizes an upper bound on application makespan.
  — For non-exponentially distributed failures we propose a Dynamic Programming approach that computes non-periodic checkpoint dates in a view to minimizing makespan.

  - For non-exponentially distributed failures we also propose a periodic checkpointing approach in which the period is computed based on a numerical search.
  - We perform simulation experiments assuming that failures follow Exponential or Weibull distributions, the latter being more representative of real-world failure behaviors [64, 63, 65, 27], and using failure logs from production clusters and the results demonstrate that group replication can be beneficial at large scale.
- For process replication:
  - We derive exact expressions for the *MNFTI* (Mean Number of Failures To Interruption) and the *MTTI* (Mean Time To Interruption) for arbitrary numbers of replicas assuming Exponential failures.
  - We extend these results to arbitrary failure distributions, notably obtaining closed-form solutions in the case of Weibull failures.
  - We perform simulation experiments and the results show that the choice of a good checkpointing period is no longer critical when process replication is used.
  - Based on our results, we can determine in which conditions the use of process replication is beneficial.

This chapter is organized as follows. Section 3.2 defines our models and states our key assumptions. Section 3.3 presents our results for group replication. Section 3.4 presents our results for process replication. Finally, Section 3.5 provides concluding remarks and perspectives.

3.2 Models and assumptions

We consider the execution of a tightly-coupled parallel application, or *job*, on a large-scale platform composed of p processors. We use the term processor to indicate any individually scheduled compute resource (a core, a multi-core processor, a cluster node), so that our work is agnostic to the granularity of the platform. We assume that standard checkpoint-recovery is performed (with checkpointing either at the system level or at the application level, with some checkpointing overhead involved). At most one application process (replica) runs on one processor.

The job must complete W units of (divisible) work, which can be split arbitrarily into separate *chunks*. We define the work unit so that when the job is executed on a single processor one unit of work is performed in one unit of time. The job can execute on any number $q \leq p$ processors. Defining $W(q)$ as the time required for a failure-free execution on q processors, we consider three models:

- Perfectly parallel jobs: $W(q) = W/q$.
- Generic parallel jobs: $W(q) = (1-\gamma)W/q + \gamma W$. As in Amdahl's law [43], $\gamma < 1$ is the fraction of the work that is inherently sequential.
- Numerical kernels: $W(q) = W/q + \gamma W^{2/3}/\sqrt{q}$, which is representative of a matrix product or a LU/QR factorization of size N on a 2D-processor grid, where $W = O(N^3)$. In the algorithm in [39], $q = r^2$ and each processor receives $2r$ blocks of size N^2/r^2 during the execution; γ is the platform's communication-to-computation ratio.

Each participating processor is subject to *failures* that each cause a *downtime*. We do not distinguish between soft and hard failures, with the understanding that soft failures are handled via software rejuvenation (i.e., rebooting [55, 56]) and that hard failures are handled by processor sparing, a common approach in production systems. For simplicity we assume that a downtime lasts D time units, regardless of the failure type. After a downtime the processor is fault-free and begins a new lifetime. In the absence of replication, when a processor fails, the whole execution is stopped, and all processors must recover from the previous checkpointed state. The recovery lasts the time needed to restore the last checkpoint from persistent storage. We assume coordinated checkpointing [82] so that no message logging/replay is needed for recovery. We allow failures to happen during recovery or checkpointing, but not during downtime (otherwise, the downtime could be considered part of the recovery). We assume that processor failures are independent and identically distributed (i.i.d.). This assumption is commonplace in the literature because it makes analysis more tractable. In the real world, instead, failures are bound to be correlated. Obtaining theoretical results for non-i.i.d. failures is beyond the scope of this work. One source of failure correlation is the hierarchical structure of compute platforms (each rack comprises compute nodes, each compute node comprises processors, each processor comprises cores), which leads to simultaneous failures of groups of processors.

We let $C(q)$ denote the time needed to perform a checkpoint, and $R(q)$ the time needed to perform a recovery. Assuming that the memory footprint of an application checkpoint is V bytes, with each processor holding V/q bytes, we consider two scenarios:

— Proportional overhead: $C(q) = R(q) = \alpha V/q = C/q$ for some constant α. This is representative of cases where the bandwidth of the network card/link at each processor is the I/O bottleneck.

— Constant overhead: $C(q) = R(q) = \alpha V = C$, which is representative of cases where the bandwidth to/from the resilient storage system is the I/O bottleneck.

Since we consider tightly coupled parallel jobs, all q processors operate synchronously. These processors execute the same amount of work $W(q)$ in parallel, chunk by chunk. The total time (on one processor) to execute a chunk of duration, or *size*, ω and then checkpoint it, is $\omega + C(q)$.

3.3 Group replication

Group replication consists in executing multiple application instances on different processor groups. All groups compute the same chunk simultaneously, and do so until one of them succeeds, potentially after several failed trials. Then all other groups stop executing that chunk and recover from the checkpoint stored by the successful group. All groups then attempt to compute the next chunk. Group replication can be implemented easily with no modification to the application, provided that the recovery implementation allows a group to recover immediately from a checkpoint produced by another group. Hereafter we formalize group replication as an execution protocol we call ASAP (As Soon As Possible).

We consider g groups, where each group has q processors, with $g \times q \leq p$. A group is available for execution if and only if all its q processors are available. In case of a failure at a processor in a group, the downtime of this group is a random variable $X_D(q) \geq D$. This random variable can take values strictly larger than D because while a processor in a group is experiencing a downtime, another processor in that group can experience a failure, thus prolonging the groups' downtime beyond D seconds. If a group encounters a first processor failure at time t, we say that the group is *down* between times t and $t + X_D(q)$.

ASAP proceeds in k macro-steps, with a chunk of work processed during each macro-step. More formally, during macro-step j, $1 \leq j \leq k$, each group independently attempts to execute the j-th chunk of size ω_j and to checkpoint, restarting as soon as possible in case of a failure. As soon as one of the groups succeeds, say at time t_j^{end}, all the other groups are immediately stopped, macro-step j is over, and macro-step $(j+1)$ starts (if $j < k$). The only two necessary inputs to the algorithm are (i) the number of chunks, k, and (ii) all chunk sizes, the ω_j's, chosen so that $\sum_{j=1}^{k} \omega_j = W(q)$.

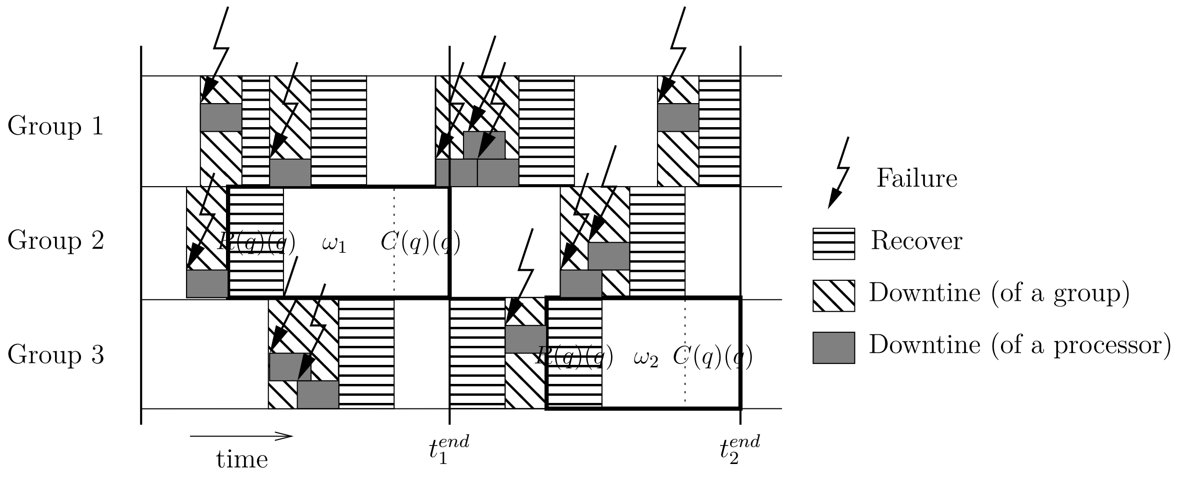


Figure 3.1: Execution of chunks ω_1 and ω_2 (macro-steps 1 and 2) using the ASAP protocol. At time t_1^{end}, Group 1 is not ready, and Group 2 is the only one that does not need to recover.

Algorithm 1: ASAP ($\omega_1, \ldots, \omega_k$)

```
1 for $j = 1$ to $k$ do
2     for each group do in parallel
3         repeat
4             Finish current downtime (if any);
5             Try to perform a recovery, then a chunk of size $\omega_j$, and finally to checkpoint;
6             if execution successful then
7                 Signal other groups to immediately stop their attempts;
8         until one of the groups has a successful attempt;
```

Before being able to start macro-step $(j+1)$, a group that has been stopped must execute a recovery so that it can resume execution from the checkpoint saved by a successful group. Furthermore, this recovery may start later than time t_j^{end}, in the case where the group is down at time t_j^{end}. This is shown on an example execution in Figure 3.1. At time t_1^{end}, Group 2 completes the computation and checkpointing of the chunk for macro-step 1. During that macro-step, Group 1 experiences two downtimes, each of duration D, while Group 3 experiences a single downtime of duration $> D$ due to a failure at a first processor followed by a failure at a second processor before the end of the first processor's downtime. At time t_1^{end}, Group 1 is down (experiencing a downtime caused by a sequence of three processor failures), so it cannot begin the recovery from the checkpoint saved by Group 2 immediately. Group 3, instead, can begin the recovery immediately a time t_1^{end}, but due to a failure it must reattempt the recovery. At time t_2^{end} it is Group 3 that completes the chunk for macro-step 2. As seen in the figure, the only groups that do not need to recover at the beginning of the next macro-step are the groups that were successful for the previous macro-step (except for the first macro-step for which all groups can start computing right away).

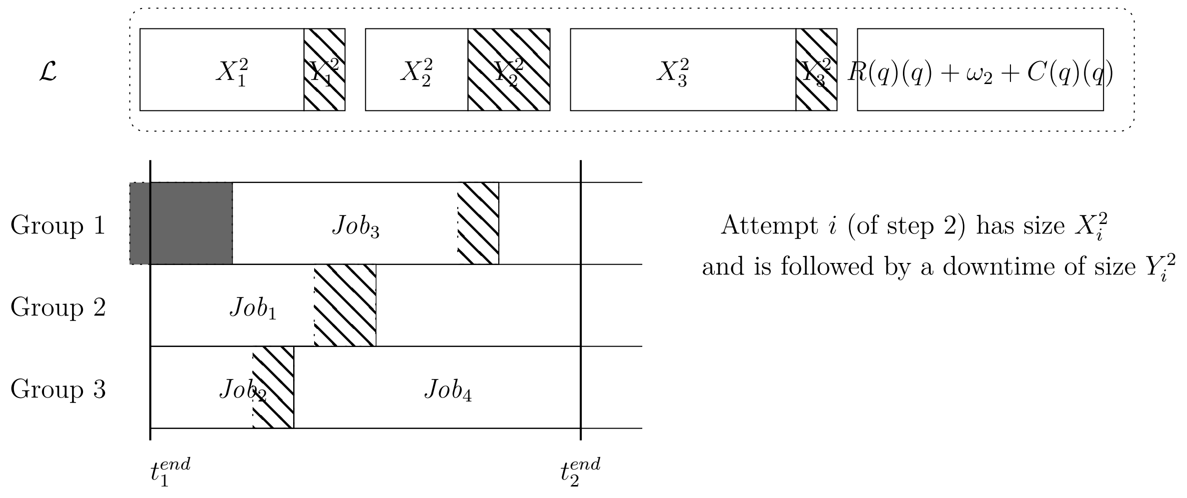


Figure 3.2: Zoom on macro-step 2 of the execution depicted in Figure 3.1, using the (X, Y) notation of Algorithm 2. Recall that Job_i has size $X_i^2 + Y_i^2$ for $1 \leq i \leq 3$, and Job_4 has size $R(q) + \omega_2 + C(q)$.

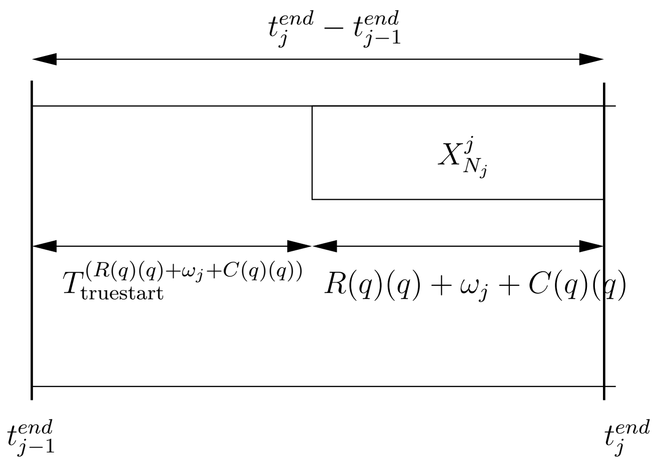
3.3.1 Exponential failures

In this section we provide an analytical evaluation of ASAP assuming Exponential failures. More specifically, we are able to compute the optimal number of macro-steps k and the optimal values of the chunk sizes ω_j. Assume that individual processor failures are distributed following an Exponential distribution of parameter λ. For the sake of the theoretical analysis, we introduce a slightly modified version of the ASAP protocol in which all groups, including the successful ones, execute a recovery at the beginning of all macro-steps, including the first one. This version of ASAP is described in Algorithm 1. It is completely symmetric, which renders its analysis easier: for macro-step j to be successful, one of the groups must be up and running for a duration of $R(q) + \omega_j + C(q)$. Note however that all experiments reported in Section 3.3.4 use the original version of ASAP, without any superfluous recovery during execution (as depicted in Figure 3.1).

Consider the j-th macro-step, number the attempts of all groups by their start time, and let N_j be the index of the earliest started attempt that successfully computes chunk ω_j. Figure 3.2 zooms in on the execution of the second macro-step ($j = 2$). Each attempt is called Job_i in the order of its start time, and is followed by a downtime but for the last attempt, which is successful. In that example the successful computation of the chunk of size $R + \omega_2 + C$ is the fourth attempt, Job_4, executed by Group 3. Consequently, $N_2 = 4$, meaning that macro-step 2 requires 4 attempts. The duration of each attempt is the sum of a sample of two random variables X_i^j and Y_i^j, $1 \leq i \leq N_j$. X_i^j corresponds to the duration of the i^{th} attempt at executing the chunk. Y_i^j corresponds to the duration of the i^{th} downtime that follows the i^{th} attempt (if $i \neq N_j$). Note that $X_i^j < R(q)+\omega_j+C(q)$ for $i < N_j$, and $X_{N_j}^j = R(q)+\omega_j+C(q)$. All the X_i^j's follow the same distribution D_X, namely an Exponential distribution of parameter $q\lambda$. And all the Y_i^j's follow the same distribution $D_{X_D}(q)$, that of the random variable $X_D(q)$ corresponding to the downtime of a group of q processors. The main idea is to view the N_j execution attempts as jobs, where the size of job i is $X_i^j + Y_i^j$, and to distribute them across the g groups using the classical online *list scheduling* algorithm for independent jobs [40, Section 5.6], as stated in the

Algorithm 2: Step j of ASAP $(\omega_1, \ldots, \omega_k)$

```
1  $i \leftarrow 1$ ;                                    /* number of attempts for the job */
2  $\mathcal{L} \leftarrow \emptyset$ ;                  /* list of attempts for the job */
3  Sample $X_i^j$ and $Y_i^j$ using $D_X$ and $D_{X_{D(q)}}$, respectively;
4  while $X_i^j < R(q) + \omega_j + C(q)$ do
5      Add $Job_i$, with processing time $X_i^j + Y_i^j$, to $\mathcal{L}$;
6      $i \leftarrow i + 1$;
7      Sample $X_i^j$ and $Y_i^j$ using $D_X$ and $D_{X_{D(q)}}$, respectively;
8  $N_j \leftarrow i$;
9  Add $Job_{N_j}$, with processing time $R(q) + \omega_j + C(q)$, to $\mathcal{L}$;
   ;        /* the first successful job has size $R(q) + \omega_j + C(q)$, not $X_{N_j}^j + Y_{N_j}^j$ */
10 From time $t_{j-1}^{end}$ on, execute a List Scheduling algorithm to distribute jobs in $\mathcal{L}$ to the different
   groups (recall that some groups may not be ready at time $t_{j-1}^{end}$);
```



$t_j^{end} - t_{j-1}^{end}$
$X_{N_j}^j$
$T_{\text{truestart}}^{(R(q)(q)+\omega_j+C(q)(q))}$
$R(q)(q) + \omega_j + C(q)(q)$
t_{j-1}^{end}
t_j^{end}

Figure 3.3: Notations used in Proposition 3.

following proposition:

Proposition 2. *The j-th ASAP macro-step can be simulated using Algorithm 2: the last job scheduled by Algorithm 2 ends exactly at time t_j^{end}.*

Proof. The List Scheduling algorithm distributes the next job to the first available group. Because of the memoryless property of Exponential laws, it is equivalent (i) to generate the attempts *a priori* and greedily schedule them, or (ii) to generate them independently within each group. ■

Proposition 3. *Let $T_{truestart}^{(R(q)+\omega_j+C(q))}$ be the time elapsed between t_{j-1}^{end} and the beginning of Job_{N_j} (see Figure 3.3). We have $\mathbb{E}\left(T_{truestart}^{(R(q)+\omega_j+C(q))}\right) \leq \mathbb{E}(Y) + \frac{\mathbb{E}(N_j)\mathbb{E}(X) - \mathbb{E}(X_j^{N_j}) + (\mathbb{E}(N_j)-1)\mathbb{E}(Y)}{g}$ where X and Y are random variables corresponding to an attempt (sampled using D_X and $D_{X_D(q)}$ respectively). Moreover, we have $\mathbb{E}(N_j) = e^{\lambda q(R(q)+\omega_j+C(q))}$ and $\mathbb{E}(X_j^{N_j}) = \frac{1}{q\lambda} + R(q) + \omega_j + C(q)$.*

Proof. For group x, $1 \leq x \leq g$, let $\tilde{Y_x}$ denote the time elapsed before it is ready for macro-step j. For example in Figure 3.2, we have $\tilde{Y_1} > 0$ (group 1 is down at time t_{j-1}^{end}), while $\tilde{Y_2} = \tilde{Y_3} = 0$ (groups 2 and 3 are ready to compute at time t_{j-1}^{end}). Proposition 2 has shown that executing macro-step j can be simulated by executing a List Schedule on a job list $\mathcal{L}$ (see Algorithm 2). We now consider g "jobs" $\tilde{Job_x}$, $x = 1, \dots, g$, so that $\tilde{Job_x}$ has duration $\tilde{Y_x}$. We now consider the augmented job list $\mathcal{L}' = \mathcal{L} \cup \bigcup_{x=1}^{g} \tilde{Job_x}$. Note that $\mathcal{L}'$ may contain more jobs than macro-step j: the jobs that start after the successful job Job_{N_j} are discarded from the list $\mathcal{L}'$. However, both schedules have the same makespan, and jobs common to both systems have the same start and completion dates. Thus, we have $T_{\text{truestart}}^{(R(q)+\omega_j+C(q))} \leq \frac{\sum_{x=1}^{g}(\tilde{Y_x}) + \sum_{i=1}^{N_j-1}(X_i^j + Y_i^j)}{g}$: this key inequality is due to the property of list scheduling: the group which is assigned the last job is the least loaded when this assignment is decided, hence its load does not exceed the average load (which is the total load divided by the number of groups). Given that $\mathbb{E}(\tilde{Y_x}) \leq \mathbb{E}(Y)$, we derive

$$\mathbb{E}\left(T_{\text{truestart}}^{(R(q)+\omega_j+C(q))}\right) \leq \mathbb{E}(Y) + \frac{\mathbb{E}\left(\sum_{i=1}^{N_j-1} X_i^j\right) + \mathbb{E}\left(\sum_{i=1}^{N_j-1}(Y_i^j)\right)}{g}$$

But N_j is the stopping criterion of the (X_i^j) sequence, hence using Wald's theorem we have $\mathbb{E}(\sum_{i=1}^{N_j} X_i^j) = \mathbb{E}(N_j)\mathbb{E}(X)$ which leads to $\mathbb{E}(\sum_{i=1}^{N_j-1} X_i^j) = \mathbb{E}(N_j)\mathbb{E}(X) - \mathbb{E}(X_j^{N_j})$. Moreover, as N_j and Y_i^j are independent variables, we have $\mathbb{E}(\sum_{i=1}^{N_j-1} Y_i^j) = (\mathbb{E}(N_j) - 1)\mathbb{E}(Y)$, and we get the desired bound for $\mathbb{E}(T_{\text{truestart}}^{(R(q)+\omega_j+C(q))})$.

Finally, as the expected number of attempts when repeating independently until success an event of probability α is $\frac{1}{\alpha}$ (geometric law), we get $\mathbb{E}(N_j) = e^{\lambda q(R(q)+\omega_j+C(q))}$. The value of $\mathbb{E}(X_j^{N_j})$ can be directly computed from the definition, recalling that $X_j^{N_j} \geq R(q) + \omega_j + C(q)$ and each X_j^i follows an Exponential distribution of parameter $q\lambda$. ■

Theorem 1. *The expected makespan of ASAP has the following upper bound:* $\frac{g-1}{g}W(q) + \frac{1}{g}\left(\frac{1}{q\lambda} + \mathbb{E}(Y)\right)e^{\lambda q(R(q)+C(q))}k^* e^{\lambda q\frac{W(q)}{k^*}} + k^*\left(\frac{g-1}{g}\left(\mathbb{E}(Y) + R(q) + C(q)\right) - \frac{1}{g}\frac{1}{q\lambda}\right)$, *where* Y *is a random variable with distribution* $D_{X_{D(q)}}$. *This bound is obtained when using* $k^* = \max(1, \lfloor k_0 \rfloor)$ *or* $k^* = \lceil k_0 \rceil$ *same-size chunks, whichever leads to the smaller value, where*

$$k_0 = \frac{\lambda q W(q)}{1 + \mathbb{L}\left(\left(g - 1 + \frac{(g-1)q\lambda(R(q)+C(q))-g}{1+q\lambda\mathbb{E}(Y)}\right)e^{-(1+\lambda q(R(q)+C(q)))}\right)}.$$

$\mathbb{L}$, *the Lambert function, is defined as* $\mathbb{L}(z)e^{\mathbb{L}(z)} = z$.

Proof. From Proposition 3, the expected execution time of ASAP has upper bound $T_{ASAP} = \sum_{j=1}^{k} \alpha_j$, where

$$\alpha_j = \mathbb{E}(Y) + \frac{\mathbb{E}(N_j)\mathbb{E}(X) - \mathbb{E}(X_j^{N_j}) + (\mathbb{E}(N_j) - 1)\mathbb{E}(Y)}{g} + (R(q) + \omega_j + C(q)).$$

Our objective now is to find the inputs to the ASAP algorithm, namely the number k of macro-steps together with the chunk sizes $(\omega_1, \dots, \omega_k)$, that minimize this T_{ASAP} bound.

We first have to prove that any optimal (in expectation) policy uses only a finite number of chunks. Let α be the expectation of the ASAP makespan using a unique chunk of size $W(q)$. According to Proposition 3,

$$\alpha = \mathbb{E}(T_{\text{truestart}}^{(R(q)+W(q)+C(q))}) + C(q) + W(q) + R(q),$$

and is finite. Thus, if an optimal policy uses k^* chunks, we must have $k^*C(q) \leq \alpha$, and thus k^* is bounded.

In the proof of Theorem 1 in [61], we have shown that any deterministic strategy uses the same sequence of chunk sizes, whatever the failure scenario, thanks to the memoryless property of the Exponential distribution. We cannot prove such a result in the current context. For instance, the number of groups performing a downtime at time t_1^{end} depends on the scenario. There is thus no reason a priori for the size of the second chunk to be independent of the scenario. To overcome this difficulty, we restrict our analysis to strategies that use the same sequence of chunk sizes whatever the failure scenario. We optimize T_{ASAP} in that context, at the possible cost of finding a larger upper bound.

We thus suppose that we have a fixed number of chunks, k, and a sequence of chunk sizes $(\omega_1, \ldots, \omega_k)$, and we look for the values of $(\omega_1, \ldots, \omega_k)$ that minimize $T_{ASAP} = \sum_{j=1}^{k} \alpha_j$. Let us first compute one of the α_j term. Replacing $\mathbb{E}(N_j)$ and $\mathbb{E}(X_j^{N_j})$ by the values given in Proposition 3, and $\mathbb{E}(X)$ by $\frac{1}{q\lambda}$, we get

$$\alpha_j = \frac{g-1}{g}\omega_j + \frac{1}{g}e^{\lambda q(R(q)+\omega_j+C(q))}\left(\frac{1}{q\lambda} + \mathbb{E}(Y)\right) + \frac{g-1}{g}\left(\mathbb{E}(Y) + R(q) + C(q)\right) - \frac{1}{g}\frac{1}{q\lambda}$$

$$T_{ASAP} = \frac{g-1}{g}W + \frac{1}{g}\left(\frac{1}{q\lambda} + \mathbb{E}(Y)\right)e^{\lambda q(R(q)+C(q))}\sum_{j=1}^{k} e^{\lambda q \omega_j} + k\left(\frac{g-1}{g}\left(\mathbb{E}(Y) + R(q) + C(q)\right) - \frac{1}{g}\frac{1}{q\lambda}\right)$$

By convexity, the expression $\sum_{j=1}^{k} e^{\lambda q \omega_j}$ is minimal when all ω_j's are equal (to $W(q)/k$). Hence all the chunks should be equal for T_{ASAP} to be minimal. We obtain:

$$T_{ASAP} = \frac{g-1}{g}W + \frac{1}{g}\left(\frac{1}{q\lambda} + \mathbb{E}(Y)\right)e^{\lambda q(R(q)+C(q))}k e^{\lambda q \frac{W(q)}{k}} + k\left(\frac{g-1}{g}\left(\mathbb{E}(Y) + R(q) + C(q)\right) - \frac{1}{g}\frac{1}{q\lambda}\right).$$

Let $f(x) = \tau_1 x e^{\lambda q \frac{W(q)}{x}} + \tau_2 x$, where

$$\tau_1 = \frac{1}{g}\left(\frac{1}{q\lambda} + \mathbb{E}(Y)\right)e^{\lambda q(R(q)+C(q))} \qquad \text{and}$$

$$\tau_2 = \left(\frac{g-1}{g}\left(\mathbb{E}(Y) + R(q) + C(q)\right) - \frac{1}{g}\frac{1}{q\lambda}\right).$$

A simple analysis using differentiation shows that f has a unique minimum, and solving $f'(x) = 0$ leads to $\tau_1 e^{\lambda q \frac{W(q)}{k}}\left(1 - \frac{\lambda q W(q)}{k}\right) + \tau_2 = 0$, and thus to $k = \frac{\lambda q W(q)}{1 + \mathbb{L}\left(\frac{\tau_2}{\tau_1 \cdot e}\right)} = k^*$, which concludes the proof. ■

This theorem can in turn be used to compute numerically the number of chunks and an upper bound on the expected makespan, provided that $\mathbb{E}(Y) = \mathbb{E}(X_D(q))$ can be itself bounded. The following proposition provides such a bound:

Proposition 4. *Let $X_D(q)$ denote the downtime of a group of q processors. Then*

$$D \le \mathbb{E}(X_D(q)) \le \frac{e^{(q-1)\lambda D} - 1}{(q-1)\lambda}. \tag{3.1}$$

Proof. In [61], we have shown that the optimal expectation of the makespan is computed as:

$$\mathbb{E}^*(q) = K^*(q)\left(\frac{1}{q\lambda} + \mathbb{E}(T_{rec}(q))\right)\left(e^{\frac{q\lambda W(q)}{K^*(q)} + q\lambda C(q)} - 1\right) \tag{3.2}$$

where $\mathbb{E}(T_{rec}(q))$ denotes the expectation of the recovery time, i.e., the time spent recovering from failure during the computation of a chunk. All chunks have the same recovery time because they all have the same size and because of the memoryless property of the Exponential distribution. It turns out that although we can compute the optimal number of chunks (and thus the chunk size), we cannot compute $\mathbb{E}^*(q)$ analytically because $\mathbb{E}(T_{rec}(q))$ is difficult to compute. We write the following recursion:

$$T_{rec}(q) = \begin{cases} X_D(q) + R(q) & \text{if no processor fails} \\ & \text{during } R(q) \text{ units of time,} \\ X_D(q) + T_{lost}(R(q)) + T_{rec}(q) & \text{otherwise.} \end{cases} \tag{3.3}$$

$X_D(q)$ is the downtime of a group of q processors, that is the time between the first failure of one of the processors and the first time at which all of them are available (accounting for the fact a processor can fail while another one is down, thus prolonging the downtime). $T_{lost}(R(q))$ is the amount of time spent computing by these processors before a first failure, knowing that the next failure occurs within the next $R(q)$ units of time. In other terms, it is the compute time that is wasted because checkpoint recovery was not completed. The time until the next failure of a group of q processors is the minimum of q *iid* Exponentially distributed variables, and is thus Exponential with parameter $q\lambda$. We can compute $\mathbb{E}(T_{lost}(R(q))) = \frac{1}{q\lambda} - \frac{R(q)}{e^{q\lambda R(q)} - 1}$ (see [61] for details). Plugging this value into Equation 3.3 leads to:

$$\begin{aligned}\mathbb{E}(T_{rec}(q)) = {} & e^{-q\lambda R(q)}(\mathbb{E}(X_D(q)) + R(q)) \\ & + (1 - e^{-q\lambda R(q)})\left(\mathbb{E}(X_D(q)) + \frac{1}{q\lambda} - \frac{R(q)}{e^{q\lambda R(q)} - 1} + \mathbb{E}(T_{rec}(q))\right)\end{aligned} \tag{3.4}$$

Equation 3.4 reads as follows: after the downtime $X_D(q)$, either the recovery succeeds for everybody, or there is a failure during the recovery and another attempt must be made. Both events are weighted by their respective probabilities. Simplifying the above expression we get:

$$\mathbb{E}(T_{rec}(q)) = \mathbb{E}(X_D(q))e^{q\lambda R(q)} + \frac{1}{q\lambda}(e^{q\lambda R(q)} - 1) \tag{3.5}$$

Plugging back this expression in Equation 3.2, we obtain the Equation:

$$\mathbb{E}^*(q) = K^*(q)\left(\frac{1}{q\lambda} + \mathbb{E}(X_D(q))\right)e^{q\lambda R(q)}\left(e^{\frac{q\lambda W(q)}{K^*(q)} + q\lambda C(q)} - 1\right) \tag{3.6}$$

Now we establish the desired bounds on $\mathbb{E}(X_D(q))$ We always have $X_D(q) \ge X_D(1) \ge D$, hence the lower bound. For the upper bound, consider a date at which one of the q processors, say processor i_0, just had a failure and initiates its downtime period for D time units. Some other processors might be in the middle of their downtime period: for each processor i, $1 \leq i \leq q$, let t_i denote the remaining duration of the downtime of processor i. We have $0 \leq t_i \leq D$ for $1 \leq i \leq q$, $t_{i_0} = D$, and $t_i = 0$ means that processor i is up and running. Let $X_D^{t_1,..,t_q}(q)$ be the *remaining* downtime of a group of q processors, knowing that processor i, $1 \leq i \leq q$, will still be down for a duration of t_i, and that a failure just happened (i.e., there exists i_0 such that $t_{i_0} = D$). Given the values of the t_i's, we have the following equation for the random variable $X_D^{t_1,..,t_q}(q)$:

$$X_D^{t_1,..,t_q}(q) = \begin{cases} D & \\ \quad \text{if none of the processors of the group} \\ \quad \text{fails during the next } D \text{ units of time} \\ T_{lost}^{t_1,..,t_q}(D) + X_D^{t'_1,..,t'_q}(q) \\ \quad \text{otherwise.} \end{cases}$$

In the second case of the equation, consider the next D time-units. Processor i can only fail in the *last* $D - t_i$ of these time-units. Here the values of the t'_i's depend on the t_i's and on $T_{lost}^{t_1,..,t_q}(D)$. Indeed, except for the last processor to fail, say i_1, for which $t'_{i_1} = D$, we have $t'_i = \max\{t'_i - T_{lost}^{t_1,..,t_q}(D), 0\}$. More importantly we always have $T_{lost}^{t_1,..,t_q}(D) \leq T_{lost}^{D,0,...,0}(D)$ and $X_D^{t_1,..,t_q}(q) \leq X_D^{D,0,..,0}(q)$ because the probability for a processor to fail during D time units is always larger than that to fail during $D - t_i$ time-units. Thus, $\mathbb{E}(X_D^{t_1,..,t_q}(q)) \leq \mathbb{E}(X_D^{D,0,..,0}(q))$. Following the same line of reasoning, we derive an upper-bound for $X_D^{D,0,..,0}(q)$:

$$X_D^{D,0,..,0}(q) \leq \begin{cases} D & \\ \quad \text{if none of the } q-1 \text{ running processors of the group} \\ \quad \text{fails during the downtime } D \\ T_{lost}^{D,0,..,0}(D) + X_D^{D,0,..,0}(q) \\ \quad \text{otherwise.} \end{cases}$$

Weighting both cases by their probability and taking expectations, we obtain

$$\mathbb{E}\left(X_D^{D,0,..,0}(q)\right) \\ \leq e^{-(q-1)\lambda D} D + (1 - e^{-(q-1)\lambda D})\left(E\left(T_{lost}^{D,0,..,0}(D)\right) + E\left(X_D^{D,0,..,0}(q)\right)\right)$$

hence $\mathbb{E}\left(X_D^{D,0,..,0}(q)\right) \leq D + (e^{(q-1)\lambda D} - 1)E\left(T_{lost}^{D,0,..,0}(D)\right)$, with

$$E\left(T_{lost}^{D,0,..,0}(D)\right) = \frac{1}{(q-1)\lambda} - \frac{D}{e^{(q-1)\lambda D} - 1}.$$

We derive

$$\mathbb{E}\left(X_D^{t_1,..,t_q}(q)\right) \leq \mathbb{E}\left(X_D^{D,..,0}(q)\right) \leq \frac{e^{(q-1)\lambda D} - 1}{(q-1)\lambda} .$$

which concludes the proof. As a sanity check, we observe that the upper bound is at least D, using the identity $e^x \geq 1 + x$ for $x \geq 0$. ■

3.3.2 General failures

The analytical derivations in Section 3.3.1 hold only for Exponential failures. In the case of non-Exponential failures we propose two algorithms for determining an execution of ASAP that achieves good makespan in practice: a "brute-force" approach called BestPeriod and a Dynamic Programming approach called DPNextFailure.

Brute-force algorithm

The BestPeriod algorithm enforces a periodic execution of ASAP, meaning that all chunk sizes are identical. For a given number of groups, the period is computed via a numerical search among a set of candidate periods generated as follow. The work in [61] makes it possible to compute an optimal period, τ, for an application executed without replication on n processors subjected to Exponential failures. In our case, with g groups and p processors, we compute this period for $n = \lfloor p/g \rfloor$ processors. Besides τ, we then generate 360 candidates as $\tau(1 + 0.05 \times i)$ and $\tau/(1 + 0.05 \times i)$ for $i \in \{1, ..., 180\}$, and 120 candidates as $\tau \times 1.1^j$ and $\tau/1.1^j$ for $j \in \{1, ..., 60\}$, for a total of 481 candidate periods. When then evaluate each candidate period in simulation (see Section 3.3.3 for details on our simulation methodology) over 50 randomly generated experimental scenarios. We pick the candidate period that achieves the best average makespan over these 50 scenarios.

BestPeriod has two potential drawbacks. First, it enforces a periodic execution even though there is no theoretical reason why the optimal should correspond to a periodic execution if failures are non-Exponential. Second, it requires running a large number of simulations ($50 \times 481 = 24,050$). With our current implementation each individual set of 481 simulations requires between 3 and 24 minutes on one core of a Quad-core AMD Opteron running at 2400 MHz. While this may indicate that BestPeriod is impractical, when compared to application makespans that can be several days the overhead of searching for the period may not be significant. Furthermore, the search for the period can be done in parallel since all simulations are independent. The search for the best period to execute an application on a large-scale platform can thus be done in a few seconds on that same large-scale platform.

Dynamic Programming algorithm

As an alternative to the brute-force algorithm in the previous section, one can resort to Dynamic Programming (DP). We initially developed a DP algorithm to compute chunk sizes for each group at each step of the application execution. Even though this seems like a natural approach, it is only tractable (in terms of number of DP states) if the chunk sizes for each group are computed independently of those for the other groups. As a result, we found that the resulting algorithm does not achieve good results in practice.

In our previous work [61], when faced with an exponential number of DP states when using DP to minimize expected makespan, we opted for maximizing the expected amount of completed work before the next failure. We generalize this idea to the context of replication, doing away with the concept of chunk sizes altogether. More specifically, since the first failure only interrupts a single group, the objective is to maximize the expected amount of work completed before all groups have failed. This can be achieved with the DP algorithm presented hereafter. We make one simplifying assumption: we ignore that once a group has failed, it will eventually restart and resume computing. This is because keeping track of such restarts would again lead to an

Algorithm 3: DPNextCheckpoint(W, T, T_0, τ_1, ..., τ_{gq})

```
1  if $W = 0$ then return $0$;
2  $best\_work \leftarrow 0$;
3  $next\_chkpt \leftarrow T$;
4  $(W_1, ..., W_g) \leftarrow$ WorkAlreadyDone($T$);          /* Work done since last recovery or
   checkpoint */
5  Sort groups by non-increasing of work done ($W_1$ is maximum);
6  for $t = T$ to $T + W - W_g$ step quantum;          /* Loop on checkpointing date */
7  do
8      $cur\_work \leftarrow 0$;
9      for $x = 1$ to $g$;  /* Loop on the first group to successfully work until $t + C(q)$ */
10     do
11         $\delta \leftarrow (t + C(q)) - T_0$;  /* Total time elapsed until the checkpoint completion */
12         $proba \leftarrow \left(\prod_{y=1}^{x-1} P_{fail}(\tau_{(y-1)q+1} + \delta, ..., \tau_{(y-1)q+q} + \delta \mid \tau_{(y-1)q+1}, ..., \tau_{(y-1)q+q})\right)$ ;
               $\times P_{suc}(\tau_{(x-1)q+1} + \delta, ..., \tau_{(x-1)q+q} + \delta \mid \tau_{(x-1)q+1}, ..., \tau_{(x-1)q+q})$
13         $\omega \leftarrow \min\{W - W_x, t - T\}$;          /* Work done between $T$ and $t$ by group $x$ */
14         $(rec\_\omega, rec\_t) \leftarrow$ DPNextCheckpoint($W - W_x - \omega$, $T + \omega + C(q) + R(q)$, $T_0$, $\tau_1, ..., \tau_{gq}$);
15         $cur\_work \leftarrow cur\_work + proba \times (W_x + \omega + rec\_\omega)$
16     if $cur\_work > best\_work$ then
17         $best\_work \leftarrow cur\_work$;
18         $next\_chkpt \leftarrow t$
19 return $(best\_work, next\_chkpt)$
```

exponential number of DP states. The hope is that our approach will work well in spite of this simplifying assumption.

Our DP algorithm, DPNextCheckpoint, is shown in Algorithm 3. It does *not* define chunk sizes, i.e., amounts of work to be processed before a checkpoint is taken, but instead it defines *checkpoint dates*. The rationale is that one checkpoint date can correspond to different amounts of work for each group, depending on when the group has started to process its chunk, after either its last failure and recovery, or its last checkpoint, or its last recovery from another group's checkpoint. Input to the algorithm is the amount of work that remains to be done (W), the current time (T), the time at which the application started (T_0), and the times since the latest failure at each processor before time T_0 (the τ_i's). The output is the next checkpoint date and the expected amount of work completed before the next failure occurs.

DPNextCheckpoint proceeds as follows. At Line 4 function WorkAlreadyDone is called which returns, for each group, the time since it has started processing its current chunk (i.e., the amount of work it has done to date). The groups are sorted in decreasing order of work performed to date (Line 5). The algorithm then picks the next checkpoint date for all possible dates between the current time T and time $T + W - W_g$, i.e., the time at which the last group would finish computing if no failure were to occur (Line 6). At the checkpointing date, the amount of work completed is the maximum of the amount of work done by the different groups that successfully complete the checkpoint. Therefore, we consider all the different cases (Line 9), that is, which group x, among the successful groups, has done the most work. We compute the probability of each case (Line 12). All groups that started to work earlier than group x have failed (i.e., at least one processor in each of them has failed) but not group x (i.e., none of its processors have failed). We compute the expectation of the amount of work completed in each case (Lines 13 and 14). We then sum the contributions of all the cases (Line 15)

Algorithm 4: DPNextFailure(W).

```
1  for each group $x = 1$ to $g$ do in parallel
2      while $W \neq 0$ do
3          $(\tau_1, ..., \tau_{gq}) \leftarrow$ Alive$(1, ..., gq)$;
4          $T_0 \leftarrow$ Time() ;                                    /* Current time */
5          $(work, date) \leftarrow$ DPNextCheckpoint$(W,\ T_0,\ T_0,\ \tau_1, \ldots, \tau_{gq})$;
6          Signal all processors that the next checkpoint date is now $date$;
7          Try to work until $date$ and then checkpoint;
8          if successful work until date and checkpoint then
9              Let $y$ be the longest running group without failure among the successful groups;
10             Let $\omega$ be the work performed by $y$ since its last recovery or checkpoint;
11             $W \leftarrow W - \omega$;
12             if group $x$'s last recovery or checkpoint was strictly later than that of $y$ then
13                 Perform a recovery;
14         if failure then Complete downtime;
15         if failure or signal then  Perform recovery from last successfully completed checkpoint
```

and record the checkpointing date leading to the largest expectation (Line 16). Note that the probability computed at Line 12 explicitly states which groups have successfully completed the checkpoint, and which groups have not. We choose not to take this information into account when computing the expectation (recursive call at Line 14) so as to avoid keeping track of which groups have failed, thereby lowering the complexity of the dynamic program. This is why the conditions do not evolve in the conditional probability at Line 12.

Algorithm 4 shows the overall algorithm, DPNextFailure, which uses DPNextCheckpoint (the Alive function returns, for a list of processors, the amount of time each has been up and running since its last downtime). Each time a group is affected by an event (a failure, a successful checkpoint by itself or by another group), it computes the next checkpoint date and broadcasts it to the g group leaders. Hence, a group may have computed the next checkpoint date to be t, and that date can be either un-modified, postponed, or advanced by events occurring at other groups and by their re-computation of the best next checkpoint date. In practice, as time is discretized, at each time quantum a group can check whether the current date is a checkpoint date or not.

Both Algorithms 3 and 4 have a complexity in $O\left(gq\left(\frac{W}{\text{quantum}}\right)^2\right)$. The term in gq comes from the computation of the probabilities at Line 12 in Algorithm 3. This complexity can be lowered using the methodology outlined in [61].

3.3.3 Simulation methodology

In this section we detail our simulation methodology. Source codes and simulation scenarios are publicly available at `http://perso.ens-lyon.fr/frederic.vivien/Data/Resilience/Replication`.

Evaluated algorithms

Our simulator implements two versions of the ASAP protocol in the case of exponentially distributed failures. The first version, OptExp, simply uses for each group the optimal and periodic policy established in [61] for Exponential failure distributions and no replication. To use OptExp with g groups we use the period from [61] computed with $\lfloor p/g \rfloor$ processors. The second, OptExpGroup, uses the periodic policy defined by Theorem 1. Both OptExp and OptExpGroup compute the checkpointing period based solely on the MTBF, assuming that failures are exponentially distributed. We nevertheless include them in all our experiments, simply using the MTBF value even when failures are not exponentially distributed. The simulator also implements BestPeriod (Section 3.3.2) and DPNextFailure (Section 3.3.2). Note that the execution times reported when using DPNextFailure include the time needed to run Algorithms 3 and 4. Based on the results in [61], we do not consider any additional checkpointing policy, such as those defined by Young [53] or Daly [54] for instance.

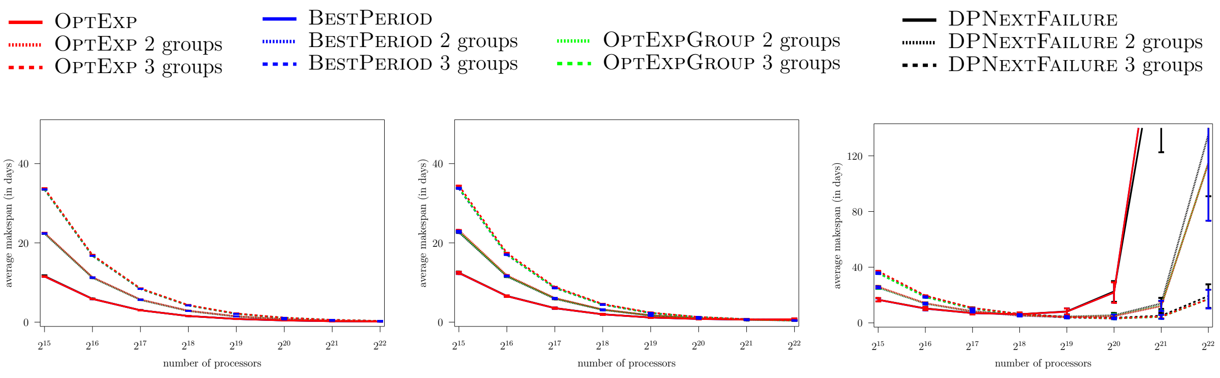
Platform and job parameters

We target two types of platforms, depending on the type of the failure distribution. For synthetic distributions, we consider platforms containing from 32,768 to 4,194,304 processors. For platforms with failures based on failure logs from production clusters, because of the limited scale of those clusters, we restrict the size of the platforms to a maximum of 131,072 processors, starting with 4,096 processors. For both platform types, we determine the job size W so that a job using the whole platform would use it for a significant amount of time in the absence of failures, namely $\approx$ 21 hours on the largest platforms for synthetic failures ($W = 10,000$ years), and $\approx$ 2.8 days on those for log-based failures ($W = 1,000$ years). In experiments with synthetic failures we use $D = 60\ s$, and $C = R = 60\ s$, $600\ s$, and $6000\ s$, thus spanning the spectrum from relatively fast to relatively slow checkpointing/recovery. We also ran experiments with a very short $C = R = 6\ s$, but the results are virtually identical to those obtained with $C = R = 60\ s$ and we do not present them. In experiments with log-based failures, we use the parameters : $C = R = 600\ s$, $D = 60\ s$. Finally, for all experiments we use $\gamma = 10^{-6}$ for generic parallel jobs, and $\gamma = 0.1$ for numerical kernels (see Section 3.2).

Failure distributions

Synthetic failure distributions – To choose failure distribution parameters that are representative of realistic systems, we use failure statistics from the Jaguar platform. Jaguar contained $45,208$ processors and is said to have experienced on the order of 1 failure per day [46]. Assuming a 1-day platform MTBF leads to a processor MTBF equal to $\frac{45{,}208}{365} \approx 125$ years. We generate both Exponential and Weibull failures, the former serving as a best case yet unrealistic scenario and the latter being representative of failure behavior in production systems [64, 63, 65, 27]. For the Exponential distribution of failure inter-arrival times, we simply set $\lambda = \frac{1}{MTBF}$. For the Weibull distribution, which requires two parameters, a shape parameter k and a scale parameter λ, and has density $\frac{k}{\lambda}(\frac{x}{\lambda})^{k-1}e^{-(x/\lambda)^k}$ for $x \geq 0$, we have $\lambda = MTBF/\Gamma(1+1/k)$. Based on the results in [64, 63, 65, 27] we use $k = 0.5$ and $k = 0.7$. For small values of the shape parameter k, the Weibull distribution is far from an Exponential distribution, meaning that it is far from being memoryless.

Log-based failure distributions – We also consider failure distributions based on failure logs from production clusters. We used logs from the largest clusters among the preprocessed logs in the *Failure trace archive* [67], i.e., from clusters at the Los Alamos National Laboratory [63]. In these logs, each failure is tagged by the node —and not just the processor— on which the failure occurred. Among the 26 possible clusters, we opted for the only two clusters with more than 1,000 nodes, as we needed a sample history sufficiently large to simulate platforms with



(a) $C = R = 60$ s (b) $C = R = 600$ s (c) $C = R = 6000$ s

Figure 3.4: Average makespan vs. number of processors, Exponential failures, $MTBF = 125$ years.

more than 10,000 nodes. The two chosen logs are for clusters 18 and 19 in the archive (referred to as 7 and 8 in [63]). For each log, we record the set $\mathcal{S}$ of availability intervals. A discrete failure distribution for the simulation is then generated as follows: the conditional probability $\mathbb{P}(X \geq t \mid X \geq \tau)$ that a node stays up for a duration t, knowing that it has been up for a duration τ, is set to the ratio of the number of availability durations in $\mathcal{S}$ greater than or equal to t, over the number of availability durations in $\mathcal{S}$ greater than or equal to τ.

Generation of failure Scenarios

Given a p-processor job, a failure trace is a set of failure dates for each processor over a fixed time horizon h (set to 2 years). The job start time is assumed to be 1 year for synthetic distribution platforms, and 0.25 year for log-based distribution platforms. We use a non-null start time to avoid side-effects related to the synchronous initialization of all nodes/processors. Given the distribution of inter-arrival times at a processor, for each processor we generate a trace via independent sampling until the target time horizon is reached. Finally, the two clusters used for computing our log-based failure distributions consist of 4-processor nodes. Hence, to simulate a 131,072-processor platform we generate 32,768 failure traces, one for each four-processor node.

3.3.4 Simulation results

In this section, we only present simulation results for perfectly parallel applications under the constant overhead model (see Section 3.2). All trends and conclusions are similar regardless of the application and overhead models. All results are averages over at least 50 instances, and all graphs show one-standard-deviation error bars.

Exponential failures

Figure 3.4 shows average makespan vs. the number of processors for our algorithms, using $g = 1$, 2, or 3 groups, and assuming Exponential failures. A first observation is that many curves overlap each other: for a given g all algorithms lead to similar average makespan. For instance, for $C = R = 600$ s and $g = 2$, and taking OptExp as a reference, the relative

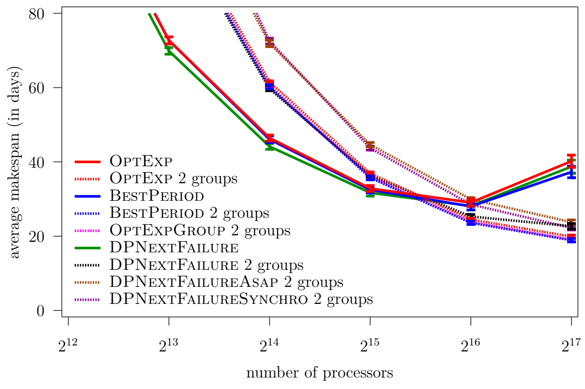


Figure 3.5: Failures based on the failure log of LANL cluster 18 ($C = R = 600$ s).

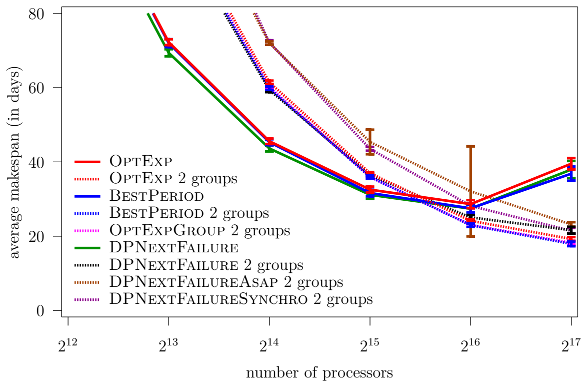
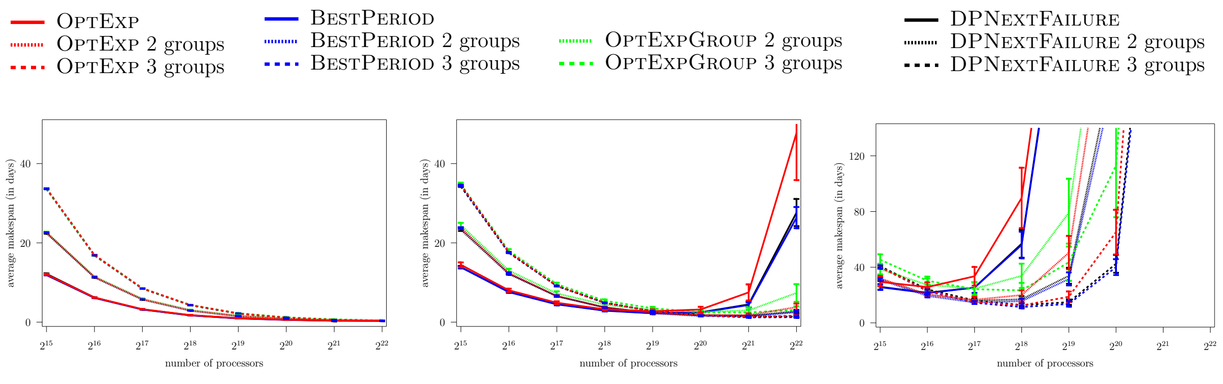


Figure 3.6: Failures based on the failure log of LANL cluster 19 ($C = R = 600$ s).

difference between the average makespan of OptExp and that of the other three algorithms is at most 6.81% (and only 2.31% when averaged over all considered numbers of processors). In spite of such small differences, several trends emerge. OptExp almost always leads to higher average makespan than OptExpGroup (note that for $g = 1$ the two algorithms are equivalent). Over the 8 numbers of processors considered, the 3 values for $R = C$, and the 3 values for g, i.e., 72 scenarios, OptExp leads to average makespans shorter than that of OptExpGroup only 4 times (for $R = C = 6000$ s, for 2^{18} to 2^{21} processors, and by at most 3.27%). BestPeriod never leads to an average makespan higher than that of OptExp or OptExpGroup, and outperforms them by up to several percent across all the $R = C$ and g values. DPNextFailure leads to mixed results, with equal or shorter average makespan than OptExpGroup, resp. BestPeriod, for 31, resp. 24, of the 72 different scenarios.

A second observation is that the use of $g > 1$ (i.e., multiple groups) often does not help and can even lead to larger average makespans. For $R = C = 60$ s, increasing g from 1 to 2, or from 2 to 3, never leads to a lower average makespan for any of our algorithms. For $R = C = 600$ s, the only improvements are seen when going from 1 to 2 groups, for the OptExp, OptExpGroup, and BestPeriod algorithms, and only with more than 2^{21} processors. The relative improvements are at most 7.75% for 2^{21} processors, and between 25.40% and 41.09% for 2^{22} processors. No improvements are achieved when going from 2 to 3 groups. More improvements are seen for $C = R = 6000$ s. When going from 1 to 2 groups, improvements are achieved starting at 2^{18} processors, with improvements up to between 93.64% and 95.17% at large scale, for all four algorithms. When going from 2 to 3 groups, relative improvements are seen starting at 2^{19} processors, reaching up to between 85.09% and 85.78% for all four algorithms.

For low and moderate checkpointing overheads, $C = R = 60$ s or 600 s, the average makespan decreases as the number of processors increases. Instead, for high checkpointing overheads, $C = R = 6000$ s, the average makespan initially decreases but starts increasing at large scale. This is particularly noticeable when using $g = 1$ group. For instance, the average makespan using OptExp goes from 21.83 s with 2^{20} processors to 249.39 s with 2^{21} processors, or an increase by a factor 11.42. The increase is similar with BestPeriod and marginally lower with DPNextFailure (a factor 9.72). The reason for this makespan increase is simply that with a high checkpointing overhead, the parallel efficiency is low as processors spend more time in checkpointing activities than in actual computation. This observation is precisely the motivation



(a) $C = R = 60$ s (b) $C = R = 600$ s (c) $C = R = 6000$ s

Figure 3.7: Average makespan vs. number of processors, Weibull failures, $k = 0.7$, $MTBF = 125$ years.

for using $g > 1$ (see Section 3.1). With $g = 2$, we still see increases in average makespans, but only by a factor between 2.46 and 2.53 when going from 2^{20} processors to 2^{21} processors for all algorithms. With $g = 3$, this factor is between 1.34 and 1.39 for all algorithms. Therefore, the use of group replication improves parallel efficiency and can lead to scalability improvements. For instance, with $g = 1$ or $g = 2$, regardless of the algorithm in use, it is not advisable to use 2^{20} processors as the makespan is lower when using 2^{19} processors. With $g = 3$, instead, there is a reduction in average makespan when going from 2^{19} processors to 2^{20} processors for all our algorithms (the relative percentage reductions are between 14.58% and 18.81%).

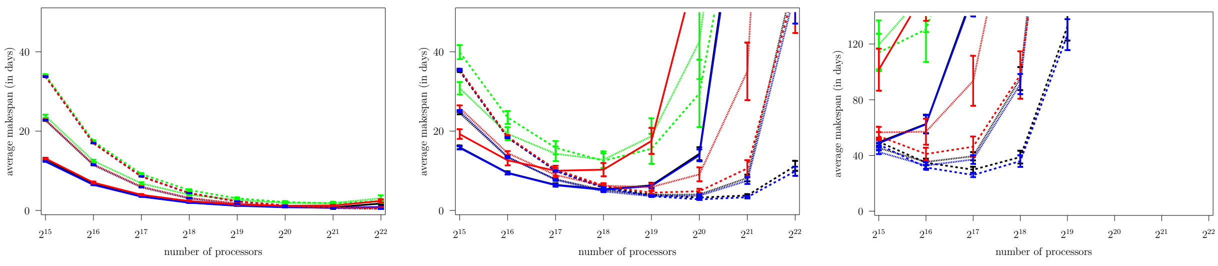
Based on the above, we conclude that for Exponential failures group replication can be useful when the checkpointing overhead is relatively large and/or when the scale of the execution is large. While large checkpointing overheads decrease parallel efficiency, the use of group replication makes it possible to limit this decrease or even to increase parallel efficiency at some scales. All our algorithms lead to comparable performance, with BestPeriod leading to good results even though marginally outperformed by DPNextFailure in some instances. While these results are interesting, and although Exponential failures have been studied in all previously published works, their relevance to practice is not clear given that real-world failures follow non-memoryless distributions. In the next section we present results for Weibull failures, which are more representative of real-world failure scenarios.

Log-based failures

For log-based failures with the constant overhead scenario, using all the available processors to run a single application instance leads to significantly larger makespans. This is seen in Figures 3.5 and 3.6, the no-replication strategies, shoot upward when p reaches a large enough value. For instance, with traces based on the logs of LANL cluster 18, the increase in makespan is more than 37% when going from $p = 2^{16}$ to $p = 2^{17}$.

Weibull failures

Figures 3.7 and 3.8 show results for Weibul failures with $k = 0.7$ and $k = 0.5$, respectively. For low $R = C = 60$ s and for $k = 0.7$ (Figure 3.3.4), results are similar to those seen in



(a) $C = R = 60$ s (b) $C = R = 600$ s (c) $C = R = 6000$ s

Figure 3.8: Average makespan vs. number of processors, Weibull failures, $k = 0.5$, $MTBF = 125$ years.

the previous section for Exponential failures: the use of multiple groups does not help, and all algorithms lead to sensibly the same performance. The gaps between the algorithms become larger for $k = 0.5$, i.e., when the failure distribution is farther from the Exponential distribution, with the advantage to BestPeriod (Figure 3.3.4). For instance, for $k = 0.5$, 2^{20} processors, and using $g = 2$ groups, BestPeriod leads to an average makespan lower than that of OptExp, OptExpGroup, and DPNextFailure by 10.46%, 51.04%, and 2.08%, respectively. A general observation in all the results for replication ($g > 1$) with Weibull failures, regardless of the value of $C = R$, is that OptExpGroup leads to much poorer results than all the other algorithms. This is because the analytical development of Theorem 1 relies heavily on the Exponential failure assumption. As a result, OptExpGroup is even outperformed by OptExp, even though this algorithm also assumes Exponential failures. In all that follows we no longer discuss the results for OptExpGroup.

For $C = R = 600$ s and $k = 0.7$, and unlike the results for Exponential failures, at large scale the average makespan of the $g = 1$ executions increases sharply while the average makespans for $g > 1$ executions remain more stable (Figure 3.3.4). In other words, even when checkpointing overheads are moderate, group replication is useful for increasing parallel efficiency once the scale is large enough. This result is amplified when failures are further from being Exponential, i.e., for $k = 0.5$ (Figure 3.3.4). For $k = 0.5$, going from $g = 1$ to $g = 2$ groups is beneficial for OptExp starting at 2^{17} processors and for BestPeriod and DPNextFailure starting at 2^{18} processors. Going from $g = 2$ to $g = 3$ groups is beneficial for OptExp andBestPeriod starting at 2^{19} processors, and for DPNextFailure starting at 2^{20} processors. In terms of comparing the algorithms with each other, in Figure 3.3.4 all algorithms experience a makespan increase after the initial decrease. Only BestPeriod and DPNextFailure, when using $g = 3$ groups, have a decreasing makespan up to 2^{20} processors. When going to 2^{21} processors, these algorithms lead to relative increases in makespan of 18.50% and 14.99%, and larger increases when going from 2^{21} to 2^{22} processors. Across the board, BestPeriod with $g = 3$ groups leads to the lowest average makespan, with DPNextFailure with $g = 3$ groups a close second. The average makespan of DPNextFailure is at most 15.66% larger than that of BestPeriod, and in fact is shorter at low scales (for 2^{15} and 2^{16} processors).

Results for $C = R = 6000$ s show similar but accentuated trends. For $k = 0.7$ (Figure 3.3.4) the main results are similar to those obtained for $k = 0.5$ with $C = R = 600$ s. The best two algorithms are BestPeriod and DPNextFailure using $g = 3$ groups, but both algorithms show an increase in makespan starting at 2^{19} processors. For $k = 0.5$ (Figure 3.3.4) this increase occurs at 2^{18} processors and is sharper for DPNEXTFAILURE than BESTPERIOD. Even though group replication helps, with such large checkpointing overheads parallel efficiency cannot be maintained beyond 2^{17} processors.

We conclude that although with Exponential failures all our algorithms are more or less equivalent (see Section 3.3.4), with more realistic Weibul failures BESTPERIOD emerges as the best algorithm. The only algorithm that leads to makespans comparable to those of BESTPERIOD is DPNEXTFAILURE, but it never leads to a lower average makespan than BESTPERIOD at large scale. Even though DPNEXTFAILURE relies on a sophisticated DP approach, the brute-force but pragmatic approach used by BESTPERIOD turns out to be more effective. Even when using BESTPERIOD, our results show that application scalability is hindered by higher checkpoint overheads, which is expected, but also by lower k values, i.e., by less exponentially distributed failures.

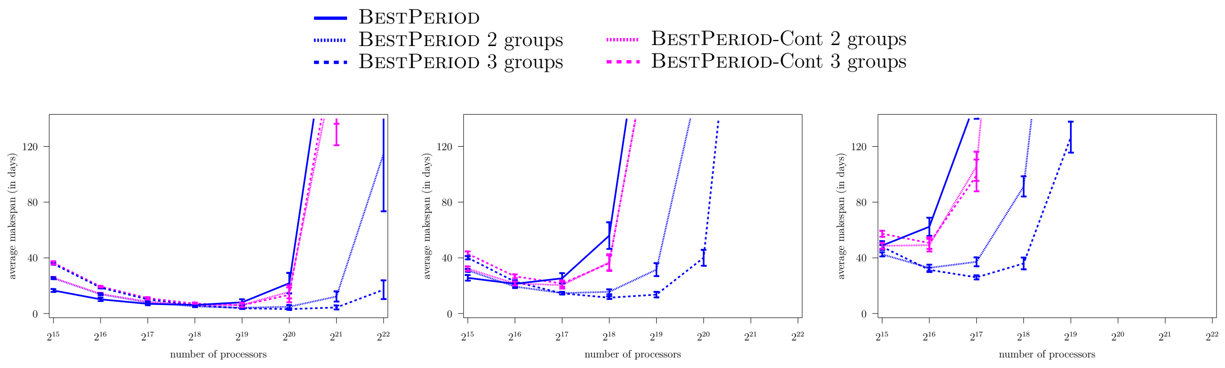
Checkpointing contention

The results presented so far are obtained assuming that the checkpointing overhead ($R = C$) does not depend on the number of groups. There are cases in which this assumption could give an unfair advantage to group replication. Consider an application with a given memory footprint V, in bytes, running on a platform with a total of q processors. With no replication ($g = 1$) the total volume of data involved in a checkpoint is V. Assuming that V is no larger than the aggregate RAM capacity of q/g processors, then group replication can be used with $g > 1$ groups. In this case, since each group executes the application, the total volume of data involved in a checkpoint at each group is also V. Since groups may checkpoint/recover at the same time, the amount of data involved can be up to $g \times V$, or a factor g larger than in the no-replication case.

To evaluate the impact of group replication on checkpointing overhead, we introduce a checkpointing contention model in our simulation. Whenever multiple checkpointing/recovery operations are concurrent, they receive a fair share of the checkpointing/recovery bandwidth. For instance, if n checkpointing operations begin at the same time, and no other checkpointing or recovery occurs over the next $n \times C$ time units, then all n checkpointing operations finish after $n \times C$ time units. More generally, considering that a checkpointing/recovery operations requires C units of activity, over a time interval Δt during which there are n ongoing such operations each operation performs $\frac{1}{n}/\Delta t$ units of activity (if one of these operations requires fewer units of work to complete, consider a shorter Δt interval).

Our objective in this section is to determine whether group replication can still be beneficial when considering checkpointing contention. We repeated all the experiments presented in Sections 3.3.4 and 3.3.4. For $C = R = 60$ s, checkpointing contention has negligible impact on the results, and the impact for $C = R = 600$ s is lower than that for $C = R = 6000$ s. This is expected since the larger the checkpointing/recovery overhead, the more likely that more than one group is engaged in checkpointing or recovery at the same time. Thus, among all our results, those for $C = R = 6000$ s should be the most disadvantageous for group replication. These are the results presented in Figure 3.9, which shows average makespan vs. number of processors for BESTPERIOD without and with contention (denoted by BESTPERIOD-Cont), for $g = 1, 2$, and 3, for $C = R = 6000$ s, for Exponential failures and for Weibull failures with $k = 0.7$ and $k = 0.5$.

As expected the average makespan of BESTPERIOD is increased due to checkpointing contention when multiple groups are used. However, even with contention, group replication out-



(a) Exponential failures (b) Weibull failures, $k = 0.7$ (c) Weibull failures, $k = 0.5$

Figure 3.9: Average makespan vs. number of processors, $C = R = 6000$ s, $MTBF = 125$ years.

performs the no-replication case at large scale. For Exponential failures, using $g = 2$ groups outperforms using $g = 1$ group as soon as the number of processors reaches 2^{18}, both with and without contention. Using $g = 3$ groups outperforms using $g = 2$ groups when there are either 2^{19} or 2^{20} processors with contention. The lowest average makespans with contention are achieved using either 2^{18} processors split in $g = 2$ groups, or 2^{19} processors split in $g = 3$ groups. For Weibull failures with $k = 0.7$, using $g = 2$ groups outperforms using $g = 1$ group starting at 2^{16} processors, with or without checkpointing contention. With contention, using $g = 3$ groups never outperforms using $g = 2$ groups, and ties its performance starting at 2^{18} processors. For Weibull failures with $k = 0.5$, using $g = 2$ groups outperforms using $g = 1$ group starting at 2^{15} processors with or without contention. With contention, using $g = 3$ groups is beneficial over using $g = 2$ groups when there are 2^{17} processors but the lowest makespan overall is achieved with $g = 2$ groups and 2^{15} processors.

We conclude that although checkpointing contention increases the makespan of group replication executions, the makespans of these executions are still shorter than that of no-replication execution at the same or slightly higher scales than when no contention takes place. One difference due to contention is that in our experiments using $g = 3$ groups is never worthwhile.

3.4 Process Replication

While in the previous section we replicate application instances, in this section we replicate processes within an instance with each each process running on a distinct processor. Process replication was recently studied in [69], in which the authors propose to replicate each application process transparently on two processors. Only when both these processors fail must the job recover from the previous checkpoint. One replica performs redundant (thus wasteful) computations, but the probability that both replicas fail is much smaller than that of a single replica, thereby allowing for a drastic reduction of checkpoint frequency.

We consider the general case where each application process is replicated $g \geq 2$ times. We call *replica-group* the set of all the replicas of a given process, and we denote by p_{total} the number of replica-groups. Altogether, if there are p available processors, there are $p_{total} \times g \leq p$ processes running on the platform. We assume that when one of the g replicas of a replica-group fails it is not restarted, and the execution of the application proceeds as long as there is still at least one running replica in each of the replica-groups. In other words, for the whole application to fail, there must exist a replica-group whose g replicas have all been "hit" by a failure. One could envision a scenario where a failed replica is restarted based on the current state of the remaining replicas in its replica-group. This would increase application resiliency but would also be time-consuming. A certain amount of time would be needed to copy the state of one of the remaining replicas. Because all replicas of a same process must have a coherent state, the execution of the still running replicas would have to be paused during this copying. In a tightly coupled application, the execution of the whole application would be paused while copying. Consequently, restarting a failed replica would only be beneficial if the restarting cost were very small, when taking in consideration the frequency of failures and the checkpoint and restart costs. The benefit of such an approach is doubtful and, like [69], we do not consider it.

3.4.1 Theoretical results

Two important quantities for evaluating the quality of an application execution, when replication is used, are: (i) the Mean Number of Failures To Interruption ($MNFTI$), i.e., the mean number of processor failures until application failure occurs; and (ii) the Mean Time To Interruption ($MTTI$), i.e., the mean time elapsed until application failure occurs. In this section, we compute exact expressions of these two quantities. We first deal with the computation of $MNFTI$ values in Section 3.4.1. Then we proceed to computing $MTTI$ values, for Exponential failures in Section 3.4.1, and for arbitrary failures in Section 3.4.1. Note that the computation of $MNFTI$ applies to any failure distribution, while that of $MTTI$ is strongly distribution-dependent.

Computing $MNFTI$

We consider two options for "counting" failures. One option is to count each failure that hits any of the $g \cdot p_{total}$ initial processors, including the processors *already hit* by a failure. Consequently, a failure that hits an already hit replica-group does not necessarily induce an application interruption. If the failure hits an already hit processor, whose replica had already been terminated due to an earlier failure, the application is not affected. If, on the contrary, the failure hits the other processor, in the case $g = 2$, then the whole application fails. This is the option chosen in [69]. Another option is to count only failures that hit *running processors*, and thus effectively kill replicas. This approach seems more natural as the running processors are the only ones that are important for the application execution.

We use $MNFTI^{\mathrm{ah}}$ to denote the $MNFTI$ with the first option ("ah" stands for "already hit"), and $MNFTI^{\mathrm{rp}}$ to denote the $MNFTI$ with the second option ("rp" stands for "running processors"). The following theorem gives a recursive expression for $MNFTI^{\mathrm{ah}}$ in the case $g = 2$ and for memoryless failure distributions.

Theorem 2. *If the failure inter-arrival times on the different processors are i.i.d. and independent from the failure history, then using process replication with $g = 2$, $MNFTI^{\mathrm{ah}} = \mathbb{E}(NFTI^{\mathrm{ah}}|0)$ where $\mathbb{E}(NFTI^{\mathrm{ah}}|n_f) =$*

$$\begin{cases} 2 & \text{if } n_f = p_{total}, \\ \frac{2p_{total}}{2p_{total}-n_f} + \frac{2p_{total}-2n_f}{2p_{total}-n_f}\mathbb{E}\left(NFTI^{\mathrm{ah}}|n_f+1\right) & \text{otherwise.} \end{cases}$$

Proof. Let $\mathbb{E}(NFTI^{\mathrm{ah}}|n_f)$ be the expectation of the number of failures needed for the whole application to fail, knowing that the application is still running and that failures have already hit n_f different replica-groups. Because each process initially has 2 replicas, this means that n_f different processes are no longer replicated, and that $p_{total} - n_f$ are still replicated. Overall, there are $n_f + 2(p_{total} - n_f) = 2p_{total} - n_f$ processors still running.

The case $n_f = p_{total}$ is the simplest. A new failure will hit an already hit replica-group, that is, a replica-group where one of the two initial replicas is still running. Two cases are then possible:

1. The failure hits the running processor. This leads to an application failure, and in this case $\mathbb{E}(\mathit{NFTI}^{\mathrm{ah}}|p_{total}) = 1$.
2. The failure hits the processor that has already been hit. Then the failure has no impact on the application. The $\mathit{MNFTI}^{\mathrm{ah}}$ of this case is then: $\mathbb{E}(\mathit{NFTI}^{\mathrm{ah}}|p_{total}) = 1 + \mathbb{E}\left(\mathit{NFTI}^{\mathrm{ah}}\,|p_{total}\right)$.

The probability of failure is uniformly distributed between the two replicas, and thus between these two cases. Weighting the values by their probabilities of occurrence yields:

$$\mathbb{E}\left(\mathit{NFTI}^{\mathrm{ah}}\,|p_{total}\right) = \frac{1}{2} \times 1 + \frac{1}{2} \times \left(1 + \mathbb{E}\left(\mathit{NFTI}^{\mathrm{ah}}\,|p_{total}\right)\right) = 2.$$

For the general case $0 \leq n_f \leq p_{total} - 1$, either the next failure hits a new replica-group, that is one with 2 replicas still running, or it hits a replica-group that has already been hit. The latter case leads to the same sub-cases as the $n_f = p_{total}$ case studied above. As we have assumed that the failure inter-arrival times on the different processors are i.i.d. and *independent from the processor failure history* the failure probability is uniformly distributed among the $2p_{total}$ processors, including the ones already hit. Hence the probability that the next failure hits a new replica-group is $\frac{2p_{total}-2n_f}{2p_{total}}$. In this case, the expected number of failures needed for the whole application to fail is one (the considered failure) plus $\mathbb{E}\left(\mathit{NFTI}^{\mathrm{ah}}|n_f+1\right)$. Altogether we have:

$$\begin{aligned}\mathbb{E}\left(\mathit{NFTI}^{\mathrm{ah}}|n_f\right) = \frac{2p_{total} - 2n_f}{2p_{total}} \times \left(1 + \mathbb{E}\left(\mathit{NFTI}^{\mathrm{ah}}|n_f+1\right)\right) \\ + \frac{2n_f}{2p_{total}} \times \left(\frac{1}{2} \times 1 + \frac{1}{2}\left(1 + \mathbb{E}\left(\mathit{NFTI}^{\mathrm{ah}}|n_f\right)\right)\right).\end{aligned}$$

Therefore, $\mathbb{E}\left(\mathit{NFTI}^{\mathrm{ah}}|n_f\right) =$

$$\frac{2p_{total}}{2p_{total}-n_f} + \frac{2p_{total}-2n_f}{2p_{total}-n_f}\mathbb{E}\left(\mathit{NFTI}^{\mathrm{ah}}|n_f+1\right). \qquad \blacksquare$$

We obtain a very similar recursive formula for $\mathit{MNFTI}^{\mathrm{rp}}$.

Theorem 3. *If the failure inter-arrival times on the different processors are independent and identically distributed, then under the process replication scheme, with $g = 2$, we have $\mathit{MNFTI}^{\mathrm{rp}} = \mathbb{E}(\mathit{NFTI}^{\mathrm{rp}}|0)$ where*

$$\mathbb{E}(\mathit{NFTI}^{\mathrm{rp}}|n_f) = \begin{cases} 1 & \text{if } n_f = p_{total}, \\ 1 + \frac{2p_{total}-2n_f}{2p_{total}-n_f}\mathbb{E}(\mathit{NFTI}^{\mathrm{rp}}|n_f+1) & \text{otherwise.}\end{cases}$$

It turns out that there is a simple (and quite unexpected) relationship between both failure models:

Proposition 5. *If the failure inter-arrival times on the different processors are i.i.d. and independent from the processor failure history then, for $g = 2$,*

$$\mathit{MNFTI}^{\mathrm{ah}} = 1 + \mathit{MNFTI}^{\mathrm{rp}}.$$

Proof. We prove by induction that $\mathbb{E}(NFTI^{\text{ah}}|n_f) = 1 + \mathbb{E}(NFTI^{\text{rp}}|n_f)$, for any $n_f \in [0, p_{total}]$. The base case is for $n_f = p_{total}$ and the induction uses non-increasing values of n_f.

For the base case, we have $\mathbb{E}(NFTI^{\text{rp}}|p_{total}) = 1$ and $\mathbb{E}(NFTI^{\text{ah}}|p_{total}) = 2$. Hence the property is true for $n_f = p_{total}$. Consider a value $n_f < p_{total}$, and assume to have proven that $\mathbb{E}(NFTI^{\text{ah}}|i) = 1 + \mathbb{E}(NFTI^{\text{rp}}|i)$, for any value of $i \in [1 + n_f, p_{total}]$. We now prove the equation for n_f. According to Theorem 2, we have:

$$\mathbb{E}(NFTI^{\text{ah}}|n_f) = \frac{2p_{total}}{2p_{total} - n_f} + \frac{2p_{total} - 2n_f}{2p_{total} - n_f}\mathbb{E}\left(NFTI^{\text{ah}}|n_f + 1\right).$$

Therefore, using the induction hypothesis, we have:

$$\begin{aligned}
&\mathbb{E}(NFTI^{\text{ah}}|n_f) \\
&= \tfrac{2p_{total}}{2p_{total}-n_f} + \tfrac{2p_{total}-2n_f}{2p_{total}-n_f}\left(1 + \mathbb{E}\left(NFTI^{\text{rp}}|n_f + 1\right)\right) \\
&= 2 + \tfrac{2p_{total}-2n_f}{2p_{total}-n_f}\mathbb{E}\left(NFTI^{\text{rp}}|n_f + 1\right) \\
&= 1 + \mathbb{E}\left(NFTI^{\text{rp}}|n_f\right)
\end{aligned}$$

the last equality being established using Theorem 3. Therefore, we have proved by induction that $\mathbb{E}(NFTI^{\text{ah}}|0) = 1 + \mathbb{E}(NFTI^{\text{rp}}|0)$. To conclude, we remark that $\mathbb{E}(NFTI^{\text{ah}}|0) = MNFTI^{\text{ah}}$ and $\mathbb{E}(NFTI^{\text{rp}}|0) = MNFTI^{\text{rp}}$. ■

We now show that Theorems 2 and 3 can be generalized to $g > 2$. Because the proofs are very similar, we only give the one for the $MNFTI^{\text{rp}}$ accounting approach (failures on running processors only), as it does not make any assumption on failures besides the i.i.d. assumption.

Proposition 6. *If the failure inter-arrival times on the different processors are i.i.d. then using process replication for $g \geq 2$, $MNFTI^{\text{rp}} = \mathbb{E}\left(NFTI^{\text{rp}}|\underbrace{0, \ldots, 0}_{g-1\ zeros}\right)$ where:*

$$\begin{aligned}
\mathbb{E}\left(NFTI^{\text{rp}}|n_f^{(1)}, \ldots, n_f^{(g-1)}\right) = 1 \\
&+ \frac{g \cdot \left(p_{total} - \sum_{i=1}^{g-1} n_f^{(i)}\right)}{g \cdot p_{total} - \sum_{i=1}^{g-1} i \cdot n_f^{(i)}} \\
&\quad \cdot \mathbb{E}\left(NFTI^{\text{rp}}|n_f^{(1)}, n_f^{(2)}, \ldots, n_f^{(g-1)}\right) \\
&+ \sum_{i=1}^{g-2} \frac{(g-i) \cdot n_f^{(i)}}{g \cdot p_{total} - \sum_{i=1}^{g-1} i \cdot n_f^{(i)}} \\
&\quad \cdot \mathbb{E}\left(NFTI^{\text{rp}}|n_f^{(1)}, \ldots, n_f^{(i-1)}, n_f^{(i)} - 1, \right. \\
&\qquad\qquad \left. n_f^{(i+1)} + 1, n_f^{(i+2)}, \ldots, n_f^{(g-1)}\right)
\end{aligned} \tag{3.7}$$

Proof. Let $\mathbb{E}\left(NFTI^{\text{rp}}|n_f^{(1)}, \ldots, n_f^{(g-1)}\right)$ be the expectation of the number of failures needed for the whole application to fail, knowing that the application is still running and that, for $i \in [1..g-1]$, there are $n_f^{(i)}$ replica-groups that have already been hit by exactly i failures. Note that a replica-group hit by i failures still contains exactly $g - i$ running replicas. Therefore, in a system where $n_f^{(i)}$ replica-groups have been hit by exactly i failures, there are still overall exactly $g \cdot p_{total} - \sum_{i=1}^{g-1} i \cdot n_f^{(i)}$ running replicas, $g \cdot \left(p_{total} - \sum_{i=1}^{g-1} n_f^{(i)}\right)$ of which are in replica-groups that have not yet been hit by any failure. Now, consider the next failure to hit the system. There are three cases to consider.

1. The failure hits a replica-group that has not been hit by any failure so far. This happens with probability:
$$\frac{g \cdot \left(p_{total} - \sum_{i=1}^{g-1} n_f^{(i)}\right)}{g \cdot p_{total} - \sum_{i=1}^{g-1} i \cdot n_f^{(i)}}$$
and, in that case, the expected number of failures needed for the whole application to fail is one (the studied failure) plus $\mathbb{E}\left(NFTI^{\text{rp}}|1 + n_f^{(1)}, n_f^{(2)}, ..., n_f^{(g-1)}\right)$. Remark that we should have conditioned the above expectation with the statement "if $p_{total} > \sum_{i=1}^{g-1} n_f^{(i)}$". In order to keep Equation (3.7) as simple as possible we rather do not explicitly state the condition and use the following abusive notation:
$$\frac{g \cdot \left(p_{total} - \sum_{i=1}^{g-1} n_f^{(i)}\right)}{g \cdot p_{total} - \sum_{i=1}^{g-1} i \cdot n_f^{(i)}} \cdot \left(1 + \mathbb{E}\left(NFTI^{\text{rp}}|1 + n_f^{(1)}, n_f^{(2)}, ..., n_f^{(g-1)}\right)\right)$$
considering than when $p_{total} = \sum_{i=1}^{g-1} n_f^{(i)}$ the first term is null and thus that it does not matter that the second term is not defined.

2. The failure hits a replica-group that has already been hit by $g-1$ failures. Such a failure leads to a failure of the whole application. As there are $n_f^{(g-1)}$ such groups, each containing exactly one running replica, this event happens with probability:
$$\frac{n_f^{(g-1)}}{g \cdot p_{total} - \sum_{i=1}^{g-1} i \cdot n_f^{(i)}}.$$
In this case, the expected number of failures needed for the whole application to fail is exactly equal to one (the considered failure).

3. The failure hits a replica-group that had already been hit by at least one failure, and by at most $g-2$ failures. Let i be any value in $[1..g-2]$. The probability that the failure hits a group that had previously been the victim of exactly i failures is equal to:
$$\frac{(g-i) \cdot n_f^{(i)}}{g \cdot p_{total} - \sum_{i=1}^{g-1} i \cdot n_f^{(i)}}$$
as there are $n_f^{(i)}$ such replica-groups and that each contains exactly $g-i$ still running replicas. In this case, the expected number of failures needed for the whole application to fail is one (the studied failure) plus $\mathbb{E}\left(NFTI^{\text{rp}}|n_f^{(1)}, ..., n_f^{(i-1)}, n_f^{(i)} - 1, n_f^{(i+1)} + 1,\right.$ $\left.n_f^{(i+2)}, ..., n_f^{(g-1)}\right)$ as there is one less replica-group hit by exactly i failures and one more hit by exactly $i+1$ failures.

We aggregate all the cases to obtain:

$$\begin{aligned}
&\mathbb{E}\left(NFTI^{\text{rp}}|n_f^{(1)},...,n_f^{(g-1)}\right) = \\
&\qquad \frac{g\cdot\left(p_{total}-\sum_{i=1}^{g-1} n_f^{(i)}\right)}{g\cdot p_{total}-\sum_{i=1}^{g-1} i\cdot n_f^{(i)}} \\
&\qquad\quad \cdot\left(1+\mathbb{E}\left(NFTI^{\text{rp}}|1+n_f^{(1)},n_f^{(2)},...,n_f^{(g-1)}\right)\right) \\
&\qquad +\sum_{i=1}^{g-2}\frac{(g-i)\cdot n_f^{(i)}}{g\cdot p_{total}-\sum_{i=1}^{g-1} i\cdot n_f^{(i)}} \\
&\qquad\quad \cdot\left(1+\mathbb{E}\left(NFTI^{\text{rp}}|n_f^{(1)},...,n_f^{(i-1)},n_f^{(i)}-1,\right.\right. \\
&\qquad\qquad\qquad \left.\left. n_f^{(i+1)}+1,n_f^{(i+2)},...,n_f^{(g-1)}\right)\right) \\
&\qquad +\frac{n_f^{(g-1)}}{g\cdot p_{total}-\sum_{i=1}^{g-1} i\cdot n_f^{(i)}}\cdot 1
\end{aligned}$$

which can be rewritten as Equation (3.7). ■

Using Proposition 6 and 5, here is the recursion to compute $MNFTI^{\text{ah}}$ for $g=3$:

Proposition 7. *If the failure inter-arrival times on the different processors are i.i.d. and independent from the failure history, then using process replication with $g=3$, $MNFTI^{\text{ah}} = \mathbb{E}(NFTI^{\text{ah}}|0,0)$ where*

$$\begin{aligned}
&\mathbb{E}\left(NFTI^{\text{ah}}|n_2,n_1\right) = \\
&\quad \frac{1}{3p_{total}-n_2-2n_1}\left(3p_{total}+3(p_{total}-n_1-n_2)\mathbb{E}\left(NFTI^{\text{ah}}|n_2+1,n_1\right)\right. \\
&\qquad\qquad \left.+2n_2\mathbb{E}\left(NFTI^{\text{ah}}|n_2-1,n_1+1\right)\right)
\end{aligned}$$

One can solve this recursion using a dynamic programming algorithm of quadratic cost $O(p^2)$ (and linear memory space $O(p)$).

Proposition 8. *If the failure inter-arrival times on the different processors are i.i.d. and independent from the failure history, then using process replication with $g=3$, $MNFTI^{\text{rp}} = \mathbb{E}(NFTI^{\text{rp}}|0,0)$ where*

$$\begin{aligned}
&\mathbb{E}\left(NFTI^{\text{rp}}|n_2,n_1\right) = \\
&\quad 1+\frac{1}{3p_{total}-n_2-2n_1}\left(3(p_{total}-n_1-n_2)\mathbb{E}\left(NFTI^{\text{rp}}|n_2+1,n_1\right)\right. \\
&\qquad\qquad \left.+2n_2\mathbb{E}\left(NFTI^{\text{rp}}|n_2-1,n_1+1\right)\right)
\end{aligned}$$

Given the simple additive relationship that exists between $MNFTI^{\text{ah}}$ and $MNFTI^{\text{rp}}$ for $g=2$ (Proposition 5), one may expect a similar relationship for large g. Table 3.1 shows $MNFTI^{\text{ah}}$ and $MNFTI^{\text{rp}}$ values and the difference between them for $g=3$. The difference is not constant and increases as p_{total} increases, and no simple relationship seems to exist between $MNFTI^{\text{ah}}$ and $MNFTI^{\text{rp}}$.

We can now evaluate our approach for computing the *MNFTI* value and compare it to that in [69]. The authors therein observe that the generalized birthday problem is related to the

Table 3.1: $MNFTI^{\text{ah}}$ and $MNFTI^{\text{rp}}$ computed using Proposition 7 and Proposition 8 and the difference between them, for $p_{total} = 2^0, \ldots, 2^{20}$, with $G = 3$.

p_{total}	2^0	2^1	2^2	2^3	2^4	2^5	2^6
$MNFTI^{\text{ah}}$	5.5	7.3	10.1	14.6	21.6	32.4	49.4
$MNFTI^{\text{rp}}$	3.0	4.5	6.9	10.9	17.1	27.1	42.9
$(MNFTI^{\text{ah}}$-$MNFTI^{\text{rp}})$	2.5	2.8	3.2	3.7	4.4	5.3	6.4

p_{total}	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}
$MNFTI^{\text{ah}}$	75.9	117.6	183.3	286.8	450.2	708.5	1117.0
$MNFTI^{\text{rp}}$	68.1	108.0	171.5	272.2	432.1	685.8	1088.7
$(MNFTI^{\text{ah}}$-$MNFTI^{\text{rp}})$	7.8	9.6	11.8	14.6	18.2	22.7	28.3

p_{total}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
$MNFTI^{\text{ah}}$	1763.5	2787.6	4410.2	6982.3	11060.6	17528.6	27788.6
$MNFTI^{\text{rp}}$	1728.1	2743.2	4354.6	6912.5	10972.9	17418.4	27650.1
$(MNFTI^{\text{ah}}$-$MNFTI^{\text{rp}})$	35.4	44.3	55.6	69.8	87.7	110.2	138.6

problem of determining the number of processor failures needed to induce an application failure. The generalized birthday problem asks the following question: what is the expected number of balls $BP(m)$ to randomly put into m (originally empty) bins so that there is a bin with two balls? This problem has a well-known closed-form solution [44]. In the context of process replication, it is tempting to consider each replica group as a bin, and each ball as a processor failure, thus computing $MNFTI = BP(p_{total})$. Unfortunately, this analogy is incorrect because processors in a replica group are distinguished. Let us consider the case $g = 2$, i.e., two replicas per replica group, and the two failure models described in Section 3.4.1. In the "already hit" model, which is used in [69], if a failure hits a replica group after that replica group has already been hit once (i.e., a second ball is placed in a bin) an application failure does not necessarily occur. This is unlike the birthday problem, in which the stopping criterion is for a bin to contain two balls, thus breaking the analogy. In the "running processor" model, the analogy also breaks down. Consider that one failure has already occurred. The replica group that has suffered that first failure is now twice less likely to be hit by another failure as all the other replica groups as it contains only one replica. Since probabilities are no longer identical across replica groups, i.e., bins, the problem is not equivalent to the generalized birthday problem. However, there is a direct and valid analogy between the process replication problem and another version of the birthday problem with distinguished types, which asks: what is the expected number of randomly drawn red or white balls $BT(m)$ to randomly put into m (originally empty) bins so that there is a bin that contains at least one red ball and one white ball? Unfortunately, there is no known closed-form formula for $BT(m)$, even though the results in Section 3.4.1 provide a recursive solution.

In spite of the above, [69] uses the solution of the generalized birthday problem to compute $MNFTI$. According to [45], a previous article by the authors of [69], it would seem that the value $BP(p_{total})$ is used. While [69] does not make it clear which value is used, a recent research report by the same authors states that they use $BP(g \cdot p_{total})$. For completeness, we include both values in the comparison hereafter.

Table 3.2 shows the $MNFTI^{\text{ah}}$ values computed as $BP(p_{total})$ or as $BP(g \cdot p_{total})$, as well as

Table 3.2: $MNFTI^{\text{ah}}$ computed as $BP(p_{total})$, $BP(g \cdot p_{total})$, and using Theorem 2, for $p_{total} = 2^0, \ldots, 2^{20}$, with $G = 2$.

p_{total}	2^0	2^1	2^2	2^3	2^4	2^5	2^6
Theorem 2	3.0	3.7	4.7	6.1	8.1	11.1	15.2
$BP(p_{total})$	2.0 (-33.3%)	2.5 (-31.8%)	3.2 (-30.9%)	4.2 (-30.3%)	5.7 (-30.0%)	7.8 (-29.7%)	10.7 (-29.6%)
$BP(g \cdot p_{total})$	2.5 (-16.7%)	3.2 (-12.2%)	4.2 (-8.8%)	5.7 (-6.4%)	7.8 (-4.6%)	10.7 (-3.3%)	14.9 (-2.3%)
p_{total}	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}
Theorem 2	21.1	29.4	41.1	57.7	81.2	114.4	161.4
$BP(p_{total})$	14.9 (-29.5%)	20.7 (-29.4%)	29.0 (-29.4%)	40.8 (-29.4%)	57.4 (-29.3%)	80.9 (-29.3%)	114.1 (-29.3%)
$BP(g \cdot p_{total})$	20.7 (-1.6%)	29.0 (-1.2%)	40.8 (-0.8%)	57.4 (-0.6%)	80.9 (-0.4%)	114.1 (-0.3%)	161.1 (-0.2%)
p_{total}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
Theorem 2	227.9	321.8	454.7	642.7	908.5	1284.4	1816.0
$BP(p_{total})$	161.1 (-29.3%)	227.5 (-29.3%)	321.5 (-29.3%)	454.4 (-29.3%)	642.4 (-29.3%)	908.2 (-29.3%)	1284.1 (-29.3%)
$BP(g \cdot p_{total})$	227.5 (-0.1%)	321.5 (-0.1%)	454.4 (-0.1%)	642.4 (-0.1%)	908.2 (-0.04%)	1284.1 (-0.03%)	1815.7 (-0.02%)

the exact value computed using Theorem 2, for various values of p_{total} and for $G = 2$. (Recall that in this case, $MNFTI^{\text{ah}}$ and $MNFTI^{\text{rp}}$ differ only by 1). The percentage relative differences between the two BP values and the exact value are included in the table as well. We see that the $BP(p_{total})$ value leads to relative differences with the exact value between 29% and 33%. This large difference seems easily explained due to the broken analogy with the generalized birthday problem. The unexpected result is that the relative difference between the $BP(g \cdot p_{total})$ value and the exact value is below 16% and, more importantly, decreases and approaches zero as p_{total} increases. The implication is that using $BP(g \cdot p_{total})$ is an effective heuristic for computing $MNFTI^{\text{ah}}$ even though the birthday problem is not analogous to the process replication problem! These results thus provide an empirical, if not theoretical, justification for the approach in [69], whose validity was not assessed experimentally therein.

Computing $MTTI$ for Exponential failures

With the "already hit" assumption, and assuming Exponential failures, the $MTTI$ can be computed easily as

$$MTTI = systemMTBF(G \times p_{total}) \times MNFTI^{\text{ah}} \tag{3.8}$$

where $systemMTBF(p)$ denotes the mean time between failures of a platform with p processors and $MNFTI^{\text{ah}}$ is given by Theorem 2. Recall that $systemMTBF(p)$ is simply equal to the MTBF of an individual processor divided by p. A recursive expression for $MTTI$ can also be obtained directly. While the $MTTI$ value should not depend on the way to count failures, it would be interesting for compute it with the "running processor" assumption as a sanity check. It turns out that there is no equivalent to Equation (3.8) for linking $MTTI$ and $MNFTI^{\text{rp}}$. The reason is straightforward. While $systemMTBF(2p_{total})$ is the expectation of the date at which the first failure will happen, it is not the expectation of the inter-arrival time of the first and second failures when only considering failures on processors still running. Indeed, after the first failure, there only remain $2p_{total} - 1$ running processors. Therefore, the inter-arrival time of the first and second failures has an expectation of $systemMTBF(2p_{total} - 1)$. We can, however, use a reasoning similar to that in the proof of Theorem 3 and obtain a recursive expression for $MTTI$:

Theorem 4. *If the failure inter-arrival times on the different processors follow an Exponential distribution of parameter λ then, when using process replication with $g = 2$, $MTTI = \mathbb{E}(TTI|0)$*

where $\mathbb{E}(TTI|n_f) =$

$$\begin{cases} \frac{1}{p_{total}}\frac{1}{\lambda} & \text{if } n_f = p_{total} \\ \frac{1}{(2p_{total}-n_f)}\frac{1}{\lambda} + \frac{2p_{total}-2n_f}{2p_{total}-n_f}\mathbb{E}(TTI|n_f+1) & \text{otherwise} \end{cases}$$

Proof. We denote by $\mathbb{E}(TTI|n_f)$ the expectation of the time an application will run before failing, knowing that the application is still running and that failures have already hit n_f different replica-groups. Since each process initially has 2 replicas, this means that n_f different processes are no longer replicated and that $p_{total} - n_f$ are still replicated. Overall, there are thus still $n_f + 2(p_{total} - n_f) = 2p_{total} - n_f$ running processors.

The case $n_f = p_{total}$ is the simplest: a new failure will hit an already hit replica-group and hence leads to an application failure. As there are exactly p_{total} remaining running processors, the inter-arrival times of the p_{total}-th and $(p_{total} + 1)$-th failures is equal to $\frac{1}{\lambda p_{total}}$ (minimum of p_{total} Exponential laws). Hence:

$$\mathbb{E}\left(TTI\,|p_{total}\right) = \frac{1}{\lambda p_{total}}.$$

For the general case, $0 \leq n_f \leq p_{total} - 1$, either the next failure hits a replica-group with still 2 running processors, or it strikes a replica-group that had already been victim of a failure. The latter case leads to an application failure; then, after n_f failures, the expected application running time before failure is equal to the inter-arrival times of the n_f-th and $(n_f + 1)$-th failures, which is equal to $\frac{1}{(2p_{total}-n_f)\lambda}$. The failure probability is uniformly distributed among the $2p_{total} - n_f$ running processors, hence the probability that the next failure strikes a new replica-group is $\frac{2p_{total}-2n_f}{2p_{total}-n_f}$. In this case, the expected application running time before failure is equal to the inter-arrival times of the n_f-th and $(n_f + 1)$-th failures plus $\mathbb{E}\left(TTI|n_f + 1\right)$. We derive that:

$$\mathbb{E}\left(TTI|n_f\right) = \frac{2p_{total} - 2n_f}{2p_{total} - n_f} \times \left(\frac{1}{(2p_{total} - n_f)\lambda} + \mathbb{E}\left(TTI|n_f + 1\right)\right) + \frac{n_f}{2p_{total} - n_f} \times \frac{1}{(2p_{total} - n_f)\lambda}.$$

Therefore,

$$\mathbb{E}\left(TTI|n_f\right) = \frac{1}{(2p_{total} - n_f)\lambda} + \frac{2p_{total} - 2n_f}{2p_{total} - n_f}\mathbb{E}\left(TTI|n_f + 1\right).$$

■

The above results can be generalized to $g \geq 2$. To compute *MTTI* under the "already hit" assumption one can use Equation (3.8) replacing $NF(p_{total})$ by the $MNFTI^{\text{ah}}$ value given by Theorem 2. To compute $MNFTI^{\text{rp}}$ under the "running processors," Theorem 4 can be generalized using the same proof technique as when proving Proposition 6.

The linear relationship between *MNFTI* and *MTTI*, seen in Equation (3.8), allows us to use the results in Table 3.2 to compute *MTTI* values. To quantify the potential benefit of

Table 3.3: *MTTI* values achieved for Exponential failures and a given number of processors using different replication factors (total of $p = 2^0, \dots, 2^{20}$ processors, with $G = 1$, 2, and 3). The individual processor MTBF is 125 years, and MTTIs are expressed in hours.

p	2^0	2^1	2^2	2^3	2^4	2^5	2^6
$g = 1$	1 095 000	547 500	273 750	136 875	68 438	34 219	17 109
$g = 2$		1 642 500	1 003 750	637 446	416 932	278 726	189 328
$g = 3$			1 505 625	999 188	778 673	565 429	432 102
p	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}
$g = 1$	8555	4277	2139	1069	535	267	134
$g = 2$	130 094	90 135	62 819	43 967	30 864	21 712	15 297
$g = 3$	326 569	251 589	194 129	151 058	117 905	92 417	72 612
p	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
$g = 1$	66.8	33.4	16.7	8.35	4.18	2.09	1.04
$g = 2$	10 789	7615	5378	3799	2685	1897	1341
$g = 3$	57 185	45 106	35 628	28 169	22 290	17 649	13 982

replication, Table 3.3 shows these values as the total number of processors increases. For a given total number of processors, we show results for $g = 1$, 2, and 3. As a safety check, we have compared these predicted values with those computed through simulations, using an individual processor MTBF equal to 125 years. For each value of p_{total} in Table 3.3, we have generated $1,000,000$ random failure dates, computed the Time To application Interruption for each instance, and computed the mean of these values. This *simulated MTTI*, is in full agreement with the predicted *MTTI* in Table 3.3

The main and expected observation in Table 3.3 is that increasing g, i.e., the level of replication, leads to increased *MTTI*. The improvement in *MTTI* due to replication increases as p_{total} increases, and increases when the level of replication, g, increases. Using $g = 2$ leads to large improvement over using $g = 1$, with an *MTTI* up to 3 orders of magnitude larger for $p_{total} = 2^{20}$. Increasing the replication level to $g = 3$ leads to more moderate improvement over $g = 2$, with an *MTTI* only about 10 times larger for $p_{total} = 2^{20}$. Overall, these results show that, at least in terms of *MTTI*, replication is beneficial. Although these results are for a particular MTBF value, they lead us to believe that moderate replication levels, namely $g = 2$, are sufficient to achieve drastic improvements in fault-tolerance.

Computing *MTTI* for arbitrary failures

The approach that computes *MTTI* from $MNFTI^{\text{ah}}$ is limited to memoryless (i.e., Exponential) failure distributions. To encompass arbitrary distributions, we use another approach based on the failure distribution density at the platform level. Theorem 5 quantifies the probability of successfully completing an amount of work of size W when using process replication for any failure distribution, which makes it possible to compute *MTTI* via numerical integration:

Theorem 5. *Consider an application with p_{total} processes, each replicated g times using process replication, so that processor P_i, $1 \leq i \leq g \cdot p_{total}$, executes a replica of process $\left\lceil \frac{i}{g} \right\rceil$. Assume that the failure inter-arrival times on the different processors are i.i.d, and let τ_i denote the time elapsed since the last failure of processor P_i. Let F denote the cumulative distribution function of the failure probability, and $F(t|\tau)$ be the probability that a processor fails in the next t units* *of time, knowing that its last failure happened τ units of time ago. Then the probability that the application will still be running after t units of time is:*

$$R(t) = \prod_{j=1}^{p_{total}} \left(1 - \prod_{i=1}^{g} F\left(t|\tau_{i+g(j-1)}\right)\right), \tag{3.9}$$

Let f denote the probability density function of the entire platform (f is the derivative of the function $1 - R$): the $MTTI$ is given by:

$$MTTI = \int_0^{+\infty} t f(t)dt = \int_0^{+\infty} R(t)dt = \int_0^{+\infty} \prod_{j=1}^{p_{total}} \left(1 - \prod_{i=1}^{g} F\left(t|\tau_{i+g(j-1)}\right)\right) \mathrm{d}t. \tag{3.10}$$

This theorem can then be used to obtain a closed-form expression for $MTTI$ when the failure distribution is Exponential (Theorem 6) or Weibull (Theorem 7):

Theorem 6. *Consider an application with p_{total} processes, each replicated g times using process replication. If the probability distribution of the time to failure of each processor is Exponential with parameter λ, then the $MTTI$ is given by:*

$$MTTI = \frac{1}{\lambda} \sum_{i=1}^{p_{total}} \sum_{j=1}^{i \cdot g} \left(\frac{\binom{p_{total}}{i}\binom{i \cdot g}{j}(-1)^{i+j}}{j} \right).$$

The following corollary gives a simpler expression for the case $g = 2$:

Corollary 1. *Consider an application with p_{total} processes, each replicated 2 times using process replication. If the probability distribution of the time to failure of each processor is Exponential with parameter λ, then the $MTTI$ is given by:*

$$\begin{aligned} MTTI &= \frac{1}{\lambda} \sum_{i=1}^{p_{total}} \sum_{j=1}^{i \cdot 2} \left(\frac{\binom{p_{total}}{i}\binom{i \cdot 2}{j}(-1)^{i+j}}{j} \right) \\ &= \frac{2^{p_{total}}}{\lambda} \sum_{i=0}^{p_{total}} \left(\frac{-1}{2}\right)^i \frac{\binom{p_{total}}{i}}{(p_{total} + i)}. \end{aligned}$$

Theorem 7. *Consider an application with p_{total} processes, each replicated g times using process replication. If the probability distribution of the time to failure of each processor is Weibull with scale parameter λ and shape parameter k, then the $MTTI$ is given by:*

$$MTTI = \frac{\lambda}{k} \Gamma\left(\frac{1}{k}\right) \sum_{i=1}^{p_{total}} \sum_{j=1}^{i \cdot g} \frac{\binom{p_{total}}{i}\binom{i \cdot g}{j}(-1)^{i+j}}{j^{\frac{1}{k}}}.$$

While Theorem 6 is yet another approach to computing the $MTTI$ for Exponential distributions, Theorem 7 is the first analytical result (to the best of our knowledge) for Weibull distributions. Unfortunately, the formula in Theorem 7 is not numerically stable for large values of p_{total}. As a result, we resort to simulation to compute $MTTI$ values. Table 3.4, which is the counterpart of Table 3.3 for Weibull failures, show $MTTI$ results obtained as averages computed on the first $100,000$ application failures of each simulated scenario. The results are similar to those in Table 3.3. The $MTTI$ with with $g = 2$ is much larger than that using $g = 1$, up to more than 3 orders of magnitude at large scale ($p_{total} = 2^{20}$). The improvement in $MTTI$ with $g = 3$ compared to $g = 2$ is more modest, reaching about a factor 10. The conclusions are thus similar: replication leads to large improvements, and a moderate replication level ($g = 2$) may be sufficient.

Table 3.4: Simulated *MTTI* values achieved for Weibull failures with shape parameter 0.7 and a given number of processors p using different replication factors (total of $p = 2^0, \ldots, 2^{20}$ processors, with $G = 1$, 2, and 3). The individual processor MTBF is 125 years, and *MTTIs* are expressed in hours.

p	2^0	2^1	2^2	2^3	2^4	2^5	2^6
$g = 1$	1 091 886	549 031	274 641	137 094	68 812	34 383	17 202
$g = 2$		2 081 689	1 243 285	769 561	491 916	321 977	214 795
$g = 3$			2 810 359	1 811 739	1 083 009	763 629	539 190
p	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}
$g = 1$	8603	4275	2132	1060	525	260	127
$g = 2$	144 359	98 660	67 768	46 764	32 520	22 496	15 767
$g = 3$	398 410	296 301	223 701	170 369	131 212	101 330	78 675
p	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
$g = 1$	60.1	27.9	12.2	5.09	2.01	0.779	0.295
$g = 2$	11 055	7766	5448	3843	2708	1906	1345
$g = 3$	61 202	47 883	37 558	29 436	23 145	18 249	14 391

3.4.2 Empirical evaluation

In the previous section, we have obtained exact expressions for the *MNFTI* and *MTTI* quantities, which are of direct relevance to the performance of the application and are amenable to analytical derivations. The main performance metric of interest to end-users, however, is the application *makespan*, i.e., the time elapsed between the launching of the application and its successful completion. But since it is not tractable to derive a closed-form expression of the expected makespan, in this section we compute the makespan empirically via simulation experiments. One of our goals here is to verify that the performance advantage of process replication seen in Sections 3.4.1 and 3.4.1 in terms of *MTTI* are also seen when considering the makespan.

Simulation framework and models

In this section we provide details on our simulation methodology for evaluating the benefits of process replication. The source code and all simulation results are publicly available at `http://graal.ens-lyon.fr/~fvivien/DATA/ProcessReplication.tar.bz2`.

Failure distributions and failure scenarios – We use the methodology described in Section 3.3.3 to generate synthetic and log-based failure distributions, the methodology described in Section 3.3.3 to generate failure scenarios,

Checkpointing policy – Replication dramatically reduces the number of application failures, so that standard periodic checkpointing strategies can be used. The checkpointing period can be computed based on the *MTTI* value using Young's approximation [53] or Daly's first-order approximation [54], the latter being used in [69]. We use Daly's approximation in this work because it is classical, often used in practice, and used in previous work [69]. It would be also interesting to present results obtained with the optimal checkpointing period, so as to evaluate the impact of the choice of the checkpointing period on our results. However, deriving the optimal period is not tractable. However, since our experiments are in simulation, we can search numerically for the best period among a sensible set of candidate periods. To build the candidate periods, we use the period computed in [61] (called OptExp) as a starting point. We then multiply and divide this period by $1 + 0.05 \times i$ with $i \in \{1, ..., 180\}$, and by 1.1^j with $j \in \{1, ..., 60\}$ and pick among these the value that leads to the lowest makespan. For a given replication level ($g{=}x$), we present results with the period computed using Daly's approximation (Daly-$g{=}x$) and with the best candidate period found numerically (BestPeriod-$g{=}x$).

Replication overhead – In [69], the authors consider that the communication overhead due to replication is proportional to the application's communication demands. Arguing that, to be scalable, an application must have sub-linear communication costs with respect to increasing processor counts, they consider an approximate logarithmic model for the percentage replication overhead: $\frac{log(p)}{10} + 3.67$, where p is the number of processors. The parameters to this model are instantiated from the application in [69] that has the highest replication overhead. When $g = 2$, we use the same logarithmic model to augment our first two parallel job models in Section 3.2:

— **Perfectly parallel jobs**: $W(p) = \frac{W}{p} \times (1 + \frac{1}{100} \times (\frac{log(p)}{10} + 3.67))$.

— **Generic parallel jobs**: $W(p) = (\frac{W}{p} + \gamma W) \times (1 + \frac{1}{100} \times (\frac{log(p)}{10} + 3.67))$.

For the numerical kernel job model, we can use a more accurate overhead model that does not rely on the above logarithmic approximation. Our original model in Section 3.2 comprises a computation component and a communication component. Using replication ($g = 2$), for each point-to-point communication between two original application processes, now a communication occurs between each process pair, considering both original processors and replicas, for a total of 4 communications. We can thus simply multiply the communication component of the model by a factor 4 and obtain the augmented model:

— **Numerical kernels**: $W(p) = \frac{W}{p} + \frac{\gamma \times W^{\frac{2}{3}}}{\sqrt{p}} \times 4$.

When $g = 3$, we (somewhat arbitrarily) multiply by $9/4$ the overhead for perfectly parallel and generic parallel jobs, because the number and volume of communications are multiplied by 4 when $g = 2$ and by 9 when $g = 3$. When $g = 3$, we multiply the communication component by a factor 9 for numerical kernels.

Parameter values – We use the following default parameter values to instantiate the simulations: $C = R = 600\ s$, $D = 60\ s$ and $W = 10,000$ years (except for log-based simulations for which $W = 1,000$ years).

Choice of the checkpointing period

Our first set of experiments aims at determining whether using Daly's approximation for computing the checkpointing period, as done in [69], is a reasonable idea when replication is used. In the $g = 2$ case (two replicas per application process), we compute this period using the exact $MTTI$ expression from Corollary 1. Given a failure distribution and a parallel job model, we compute the average makespan over 100 sample simulated application executions for a range of numbers of processors. Each sample is obtained using a different seed for generating random failure events based on the failure distribution. We present results using the best period found via a numerical search in a similar manner. In addition to the $g = 2$ and $g = 3$ results, we also present results for $g = 1$ (no replication) as a baseline, in which case the $MTTI$ is simply the processor MTBF. In the three options the total number of processors is the same, i.e., $g \times n/g$.

We show experimental results for five failure distributions: (i) Exponential with a 125-year MTBF; (ii) Weibull with a 125-year MTBF and shape parameter $k = 0.70$; (iii) Weibull with a 125-year MTBF and shape parameter $k = 0.50$; (iv) Failures drawn from the failure log of

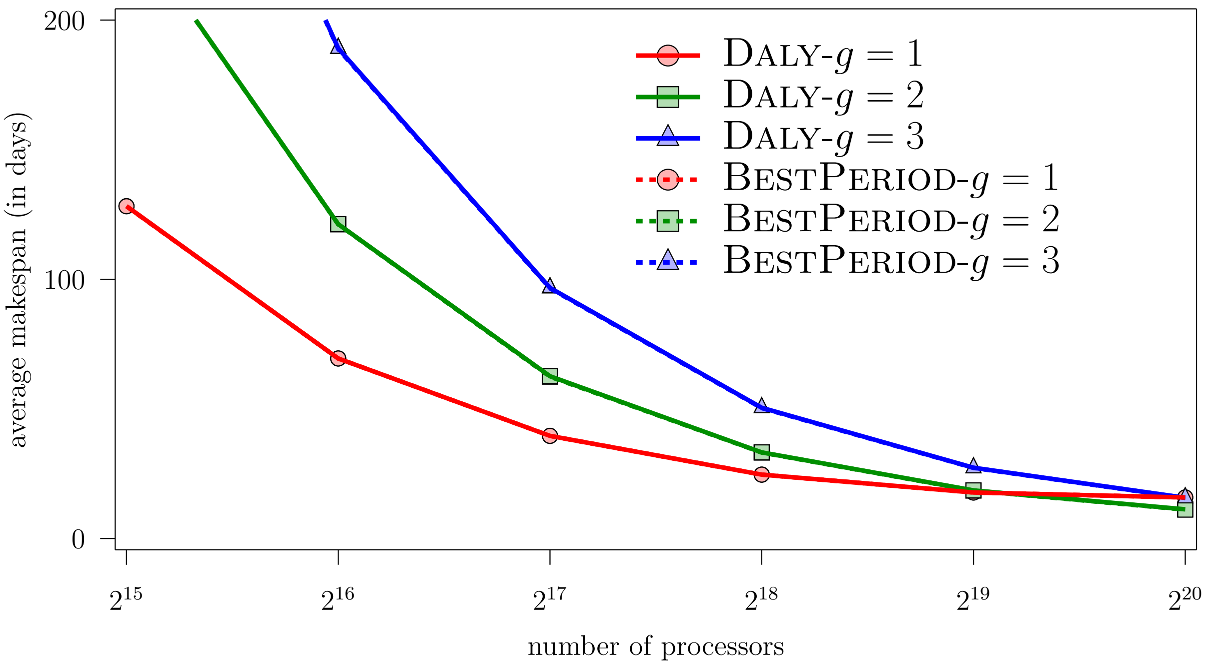


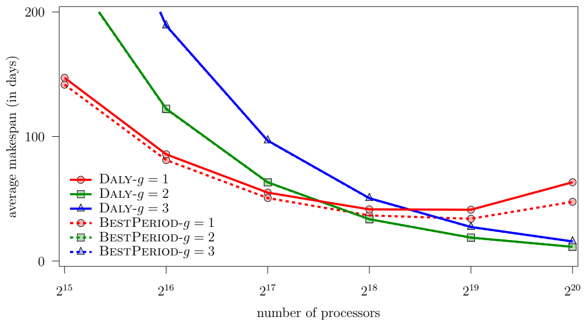
Figure 3.10: Average makespan vs. number of processors for two choices of the checkpointing period, without process replication (DALY-g=1 and BESTPERIOD-g=1) and with process replication (DALY-g=2 or 3 and BESTPERIOD-g=2 or 3), for generic parallel jobs subject to Exponential failures ($MTBF = 125$ years).

LANL cluster 18; and (v) Failures drawn from the failure log of LANL cluster 19. For each failure distribution, we use five parallel job models as described in Section 3.2, augmented with the replication overhead model described in Section 3.4.2: (i) perfectly parallel; (ii) generic parallel jobs with $\gamma = 10^{-6}$; (iii) numerical kernels with $\gamma = 0.1$; (iv) numerical kernels with $\gamma = 1$; and (v) numerical kernels with $\gamma = 10$. We thus have $5 \times 5 = 25$ sets of results.

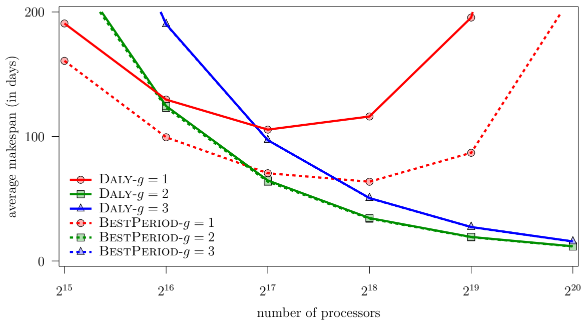
Figures 3.10 through 3.13 show average makespan vs. number of processors. It turns out that, for a given failure distribution, all results follow the same trend regardless of the job model, as illustrated in Figure 3.13 for Weibull failures with $k = 0.7$. But for Figure 3.13 we show results only for generic parallel jobs.

Figures 3.10, 3.11, and 3.12 show average makespan vs. number of processors for generic parallel jobs subject to each of the five considered failure distributions. We first note that, except for Exponential failures, the minimum makespan is not achieved on the largest platform. The fact that in most cases the makespan with 2^{19} processors is lower than the makespan with 2^{20} processors suggests that duplicating processes should be beneficial. This is indeed always the case for the largest platforms: when using 2^{20} processors, the makespan without replication is always larger than the makespan with replication, the replication factor being either $g = 2$ or $g = 3$. However, in none of the configurations, using a replication with f $g = 3$ is more beneficial than with $g = 2$. More importantly, in each configuration, the minimum makespan is always achieved while duplicating the processes ($g = 2$) and using the maximum number of processors.

The two curves for $g = 1$ are exactly superposed in Figure 3.10. For $g = 2$ and for $g = 3$ the two curves are exactly superposed in all three figures. Results for the case $g = 1$ (no replication) show that Daly's approximation achieves the same performance as the best periodic

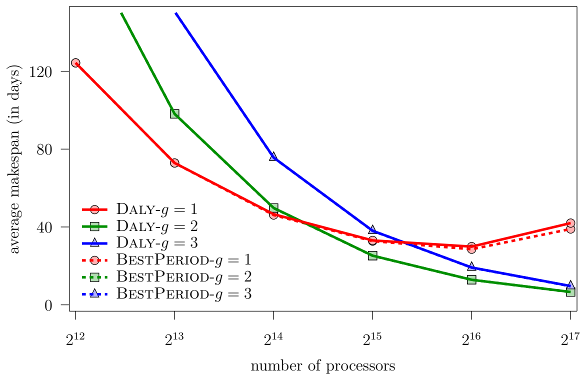


(a) $k = 0.70$

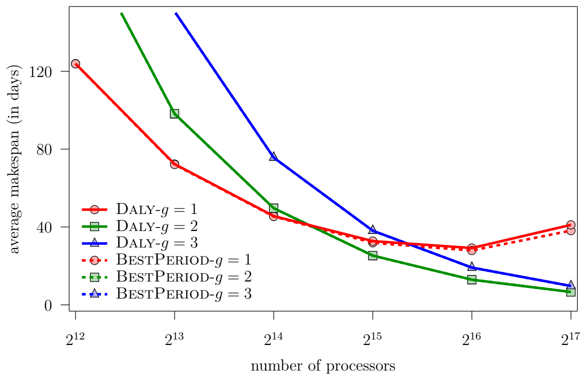


(b) $k = 0.50$

Figure 3.11: Same as Figure 3.10 (generic parallel jobs) but for Weibull failures ($MTBF = 125$ years).



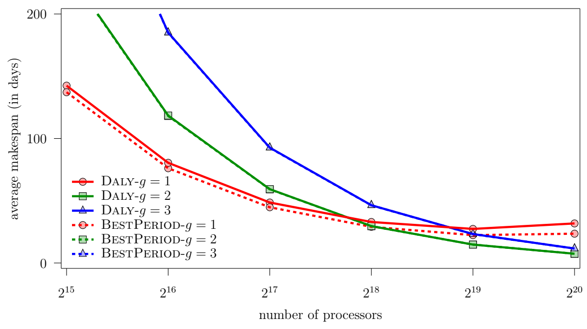
(a) LANL cluster 18



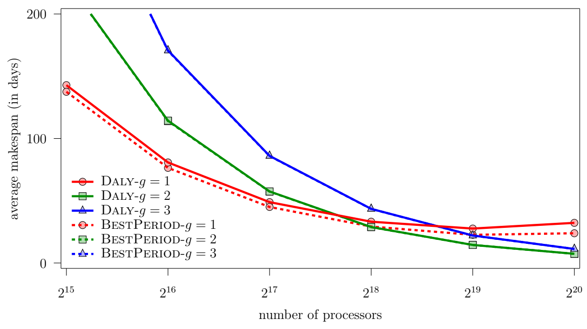
(b) LANL cluster 19

Figure 3.12: Same as Figure 3.10 (generic parallel jobs) but for real-world failures.

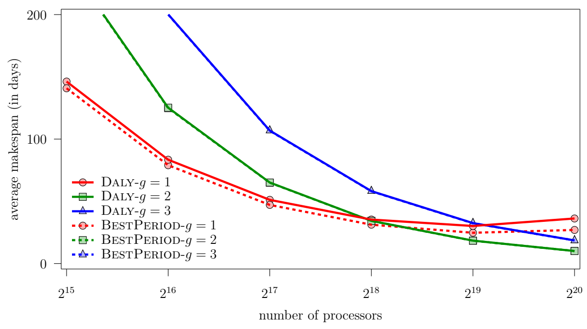
checkpointing policy for Exponential failures. For our two real-world failure datasets, using the approximation also does well, deviating from the best periodic checkpointing policy only marginally as the platform becomes large. For Weibull failures, however, Daly's approximation leads to significantly suboptimal results that worsen as k decreases (as expected and already reported in [61]). What is perhaps less expected is that in the cases $g = 2$ and $g = 3$, using Daly's approximation leads to virtually the same performance as using the best period even for Weibull failures. With replication, application makespan is simply not sensitive to the checkpointing period, at least in a wide neighborhood around the best period. This is because application failures and recoveries are infrequent, i.e., the MTBF of a pair of replicas is large. To quantify the frequency of application failures Table 3.5 shows the percentage of processor failures that actually lead to failure recoveries when using process replication. Results are shown in the case of Weibull failures for $k = 0.5$ and $k = 0.7$, $C = 600s$, and for various numbers of processors. We see that very few application failures, and thus recoveries, occur throughout application execution (recall that makespans are measured in days in our experiments). This is because a very small fraction of processor failures manifest themselves as application failures (below 0.4% in our experiments). This also explains why using $g = 3$ replicas does not lead to any further performance improvements (recall that the expectation was that further improvement would be low anyway given the results in Tables 3.3 and 3.4) . While this low number of application failures demonstrates the benefit of process replication, the interesting result is that it also



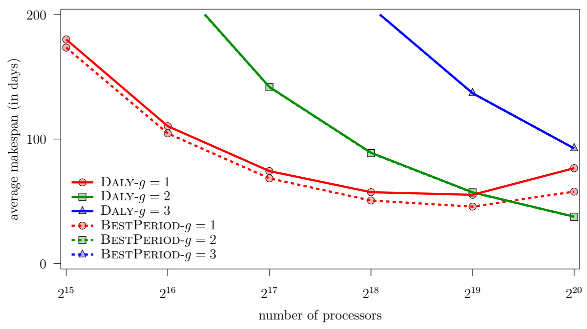
(a) Perfectly parallel jobs



(b) Numerical kernels ($\gamma = 0.1$)



(c) Numerical kernels ($\gamma = 1$)



(d) Numerical kernels ($\gamma = 10$)

Figure 3.13: Same as Figure 3.11(a) (Weibull failures with $k = 0.7$, $MTBF = 125$ years) but for other job types.

makes the choice of the checkpointing period not critical.

When setting the processor MTBF to a lower value so that the MTBF of a pair of replicas is not as large, then the choice of the checkpointing period matters. Consider for instance a process replication scenario with Weibull failures of shape parameters $k = 0.7$, a generic parallel job, and a platform with 2^{20} processors. When setting the MTBF to an unrealistic 0.1 year, using Daly's approximation yields an average makespan of 22.7 days, as opposed to 19.1 days (an increase of more than 18%) when using the best period. Similar cases can be found for Exponential failures.

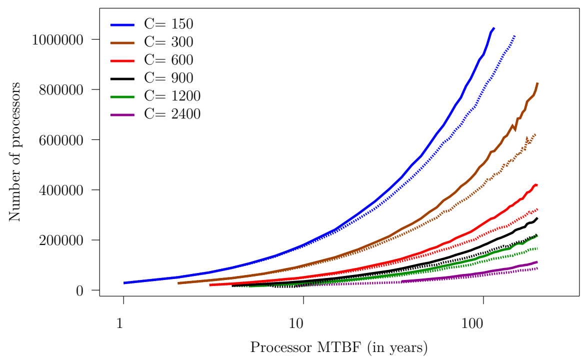
We summarize our findings so far as follows. Without replication, a poor choice of checkpointing period produces significantly suboptimal performance. When using replication, a poor choice can also theoretically lead to poor results, but this is very unlikely in practice because replication drastically reduces the number of failures. In fact, in practical settings, the choice of the checkpointing period is simply not critical when replication is used. Consequently, setting the checkpointing period based on an approximation, Daly's being the most commonplace and oft referenced, is appropriate.

When is process replication beneficial?

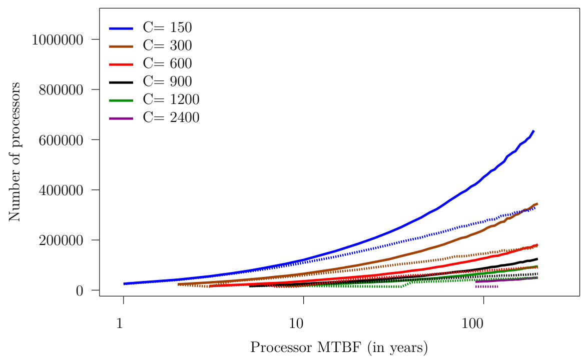
In this section we determine under which conditions process replication is beneficial, i.e., leads to a lower makespan, when compared to a standard application execution that uses only

Table 3.5: Number of application failures and fraction of processor failures that cause application failures with process replication ($g = 2$) assuming Weibull failure distributions ($k = 0.7$ or 0.5) for various numbers of processors and C=600s. Results are averaged over 100 experiments.

	# of app. failures		% of proc. failures	
# of proc.	$k = 0.7$	$k = 0.5$	$k = 0.7$	$k = 0.5$
2^{14}	1.95	4.94	0.35	0.39
2^{15}	1.44	3.77	0.25	0.28
2^{16}	0.88	2.61	0.15	0.19
2^{17}	0.45	1.67	0.075	0.12
2^{18}	0.20	1.11	0.034	0.076
2^{19}	0.13	0.72	0.022	0.049
2^{20}	0.083	0.33	0.014	0.023



(a) $k = 0.70$



(b) $k = 0.50$

Figure 3.14: Break-even point curves for replication ($g = 2$) vs. no-replication for various checkpointing overheads, as computed using the best checkpointing periods (solid lines) and Daly's approximation (dashed lines), assuming Weibull failure distributions.

checkpoint-recovery. We restrict our study to duplication ($g = 2$) as we have seen that the case $g = 3$ was never beneficial with respect to the case $g = 2$.

In a 2-D plane defined by the processor MTBF and the number of processors, and given a checkpointing overhead, simulation results can be used to construct a curve that divides the plane into two regions. Points above the curve correspond to cases in which process replication is beneficial. Points below the curve correspond to cases in which process replication is detrimental, i.e., the resource waste due to replication is not worthwhile because the processor MTBF is too large or the number of processors is too low. Several such curves are shown in [69] (Figure 9 therein) for different checkpointing overheads, and, as expected, the higher the overhead the more beneficial it is to use process replication.

One question when comparing the replication and the no-replication cases is that of the checkpointing period. We have seen in the previous section that when using process replication the choice of the period has little impact and that Daly's approximation can be used safely. In the no-replication case, however, Daly's approximation should only be used in the case of exponentially distributed failures as it leads to poor results when the failure distribution is Weibull (see the $g = 1$ curves in Figure 3.11). Although our results for two particular production workloads show that Daly's approximation leads to reasonably good results in the no-replication case (see the $g = 1$ curves in Figures 3.12), there is evidence that, in general, failure distributions are well approximated by Weibull distributions [64, 63, 65, 27], while not at all by exponential distributions. Most recently, in [27], the authors show that failures observed on a production cluster, over a cumulative 42-month time period, are modeled well by a Weibull distribution with shape parameter $k < 0.5$. In other words, the failure distribution is far from being Exponential and thus Daly's approximation would be far from the best period (compare Figure 3.11(a) for $k = 0.7$ to Figure 3.11(b) for $k = 0.5$).

Given the above, comparing the replication case to the no-replication case with Weibull failure distributions and using Daly's approximation as the checkpointing period gives an unfair advantage to process replication. To isolate the effect of replication from checkpointing period effects, we opt for the following method: we always use the best checkpointing period for each simulated application execution, as computed by a numerical search over a range of simulated executions each with a different checkpointing period. These results, for $g = 2$, are shown as solid curves in Figure 3.14, for Weibull failures with $k = 0.7$ and $k = 0.5$, each curve corresponding to a different checkpointing overhead (C) value.

Each curve corresponds to the break-even point and the area above the curve corresponds to settings for which replication is beneficial. As expected, replication becomes detrimental when the number of processors is too small, when the checkpointing overhead is too low, and/or when the processor MTBF is too large. For comparison purposes, the figure also shows a set of dashed curves that correspond to results obtained when using Daly's approximation as the checkpointing period instead of using our numerical search for the best such period. We see that, as expected, using Daly's approximation gives an unfair advantage to process replication. This advantage increases as k decreases, since the Weibull distribution is then further away from the Exponential distribution. (For exponential distributions, all curves match.) For instance, for $k = 0.5$ (Figure 3.14(b)), the break-even curve for $C = 600s$ as obtained using Daly's approximation is in fact, for most values of the MTBF, below the break-even curve for $C = 900s$ as obtained using the best checkpointing period. Note that the results presented in [69] are obtained using Daly's approximation as the checkpointing period.

3.5 Conclusion

In this chapter we have presented a rigorous study of replication techniques for large-scale platforms. These platforms are subject to failures, the frequencies of which increase dramatically with platform scale. We have investigated replication as a technique to better use all the resources provided by the platform. Replication comes in two flavors, Group replication and Process replication. Group replication consists in partitioning the platform into several groups, which each executes an instance of the application concurrently in phases. All groups synchronize as soon as one of them completes a phase. Instead, Process replication replicates each application process onto several processors (a replica-group), thereby reducing the need to recover from a failure only when all processors in a replica-group have failed. Process replication is the approach followed in [69] with two processors per replica-group.

While both replication techniques improve reliability, they have very different characteristics. Group replication can be used for any kind of parallel application, while Process replication is much more intrusive than Group replication, in that it requires a sophisticated replication-aware implementation of the MPI library. Also, the total communication volume is increased by a factor proportional to the square of the replication degree, while the increase is only linear for Group replication.

We also have provided a detailed analysis of group replication for large-scale platforms. We have defined an execution protocol for group replication, ASAP. We have derived a bound on the expected application makespan using this protocol when failures are exponentially distributed, which suggests a checkpointing period that can be used in practice. We have also proposed two approaches to minimize application makespan that are applicable regardless of the failure distribution: (i) a brute-force search for a checkpointing period, called BestPeriod; and (ii) a Dynamic Programming algorithm, called DPNextFailure. Using simulation, and for a range of failure and checkpointing overheads, we have evaluated our proposed approaches and compared them to no-replication approaches from previous work. Our main findings are that (i) when considering realistic failures (e.g., Weibull distributed) group replication can significantly lower application makespan on large-scale platforms; (ii) our pragmatic BestPeriod approach outperforms the more sophisticated DPNextFailure Dynamic Programming approach; (iii) even when accounting for the contention due to concurrent checkpointing/recovery by multiple groups, group replication remains beneficial at large scale. Note that our group replication approaches lead to particularly good results when failures are far from being exponentially distributed, which several studies have shown to be the case in production platforms [64, 63, 65, 27].

We have provided a thorough analysis of Process replication for large-scale platforms. We have obtained recursive expressions for $MNFTI$, and analytical expressions for $MTTI$ with arbitrary distributions, which lead to closed-form expressions for Exponential and Weibull distributions. We have also identified an unexpected relationship between two natural failure models (*already hit* and *running processors*) in the case of process duplication ($g = 2$).

We have conducted an extensive set of simulations for Exponential, Weibull, and trace-based failure distributions. These results have shown that although the choice of a good checkpointing period can be important in the no-replication case, namely for Weibull failure distributions, this choice is not critical when process replication is used. This is because with process replication few processor failures lead to application failures (i.e., rollback and recovery). This effect is essentially the reason why process replication was proposed in the first place. But a surprising and interesting side-effect is that choosing a good checkpointing period is no longer challenging. Finally, we have determined the break-even point between replication and no-replication for Weibull failures. Unlike that in previous work, this determination is agnostic to the choice of the checkpointing period, leading to results that are not as favorable for process replication. Our results nevertheless point to relevant scenarios, defined by instantiations of the platform and application parameters, in which replication is worthwhile when compared to the no-replication case. This is in spite of the induced resource waste, and even if the best checkpointing period is used in the no-replication case.

An interesting direction for future work on Group replication would be to compare the checkpoints saved by multiple groups as a way to detect silent errors or corrupted data. This would require modifying the Group replication approach so that at least 2 groups among $g > 2$ groups compute a chunk of work successfully, thereby trading off performance for reliability.

Chapter 4

Combining Fault Prediction and Coordinated Checkpointing

4.1 Introduction

In this Chapter, we assess the impact of fault prediction techniques on checkpointing strategies. When some fault prediction mechanism is available, can we compute a better checkpointing period to decrease the expected waste? and to what extent? Critical parameters that characterize a fault prediction system are its recall r, which is the fraction of faults that are indeed predicted, and its precision p, which is the fraction of predictions that are correct (i.e., correspond to actual faults).

The major objective of this Chapter is to refine the expression of the expected waste as a function of these new parameters, and to design efficient checkpointing policies that take predictions into account.

In this chapter, we deal with two problem instances, one where the predictor system provides *exact dates* for predicted events, and another where it only provides *prediction windows* during which events take place.The key contributions of this chapter are :

— With a Predictor with exact prediction dates :
    — The design of new checkpointing policies that takes optimal decisions on whether and when to take the predictions into account (or to ignore them).
    — For policies where the decision to trust the predictor is taken with the same probability throughout the checkpointing period, we show that we should always trust the predictor, or never, depending upon platform and predictor parameters.
    — For policies where the decision to trust the predictor is taken with variable probability during the checkpointing period, we show that we should change strategy only once in the period, moving from never trusting the predictor when the prediction arrives in the beginning of the period, to always trusting the predictor when the prediction arrives later on in the period, and we determine the optimal break-even point.
    — For all policies, we compute the optimal value of the checkpointing period thereby designing optimal algorithms to minimize the waste when coupling checkpointing with predictions.
    — An extensive set of simulations that corroborates all mathematical derivations. These simulations are based on synthetic fault traces (for Exponential fault distributions, and for more realistic Weibull fault distributions) and on log-based fault traces. In addition, they include exact prediction dates and uncertainty intervals for these dates.

Note that the subsection on uncertain intervals is thoroughly studied in Section 4.3.

— With a Predictor with a prediction window :
  — The design of several checkpointing policies that account for the different sizes of prediction windows.
  — The analytical characterization of the best policy for each set of parameters.
  — The validation of the theoretical results via extensive simulations, for both Exponential and Weibull failure distributions.

In the Table 4.1, we summarize the main notations used in this chapter.

p	Predictor precision: proportion of true positives among the number of predicted faults
r	Predictor recall: proportion of predicted faults among total number of faults
q	Probability to trust the predictor
MTBF	Mean Time Between Faults
N	Number of processors in the platform
μ	Platform MTBF
μ_{ind}	Individual MTBF
μ_{P}	Rate of predicted faults
μ_{NP}	Rate of unpredicted faults
μ_{e}	Rate of events (predictions or unpredicted faults)
D	Downtime
R	Recovery time
C	Duration of a regular checkpoint
C_p	Duration of a proactive checkpoint
T	Duration of a period

Table 4.1: Table of main notations.

4.2 Predictor with exact prediction dates

In this section, we present an analytical model to assess the impact of predictions on periodic checkpointing strategies. We consider the case where the predictor is able to provide exact prediction dates, and to generate such predictions at least C_p seconds in advance, so that a proactive checkpoint of length C_p can indeed be taken before the event.

This Section is organized as follows. We first detail the framework in subsection 4.2.1. We provide optimal algorithms to account for predictions in subsection 4.2.2: we start with simpler policies where the decision to trust the predictor is taken with the same probability throughout the checkpointing period (subsection 4.2.2) before dealing with the most general approach where the decision to trust the predictor is taken with variable probability during the checkpointing period (subsection 4.2.2). subsection 4.2.3 is devoted to simulations: we first describe the simulation framework (subsection 4.2.3) and then discuss synthetic and log-based failure traces in subsections 4.2.3 and 4.2.3 respectively. Finally, we provide concluding remarks in subsection 4.4.

4.2.1 Framework

Checkpointing strategy

We consider a *platform* subject to faults. Our work is agnostic of the granularity of the platform, which may consist either of a single processor, or of several processors that work concurrently and use coordinated checkpointing. *Checkpoints* are taken at regular intervals, or periods, of length T. We denote by C the duration of a checkpoint (all checkpoints have same duration). By construction, we must enforce that $C \leq T$. When a fault strikes the platform, the application is lacking some resource for a certain period of time of length D, the *downtime*. The downtime accounts for software rejuvenation (i.e., rebooting [55, 56]) or for the replacement of the failed hardware component by a spare one. Then, the application recovers from the last checkpoint. R denotes the duration of this *recovery* time.

Fault predictor

A fault predictor is a mechanism that is able to predict that some faults will take place, either at a certain point in time, or within some time-interval window. In this Section, we assume that the predictor is able to provide exact prediction dates, and to generate such predictions early enough so that a *proactive* checkpoint can indeed be taken before the event.

The accuracy of the fault predictor is characterized by two quantities, the *recall* and the *precision*. The recall r is the fraction of faults that are predicted while the precision p is the fraction of fault predictions that are correct. Traditionally, one defines three types of *events*: (i) *True positive* events are faults that the predictor has been able to predict (let $True_P$ be their number); (ii) *False positive* events are fault predictions that did not materialize as actual faults (let $False_P$ be their number); and (iii) *False negative* events are faults that were not predicted (let $False_N$ be their number). With these definitions, we have $r = \frac{True_P}{True_P + False_N}$ and $p = \frac{True_P}{True_P + False_P}$.

Proactive checkpoints may have a different length C_p than regular checkpoints of length C. In fact there are many scenarios. On the one hand, we may well have $C_p > C$ in scenarios where regular checkpoints are taken at time-steps where the application memory footprint is minimal [57]; on the contrary, proactive checkpoints are taken according to predictions that can take place at arbitrary instants. On the other hand, we may have $C_p < C$ in other scenarios [52], e.g., when the prediction is localized to a particular resource subset, hence allowing for a smaller volume of checkpointed data.

To keep full generality, we deal with two checkpoint sizes in this Section: C for periodic checkpoints, and C_p for proactive checkpoints (those taken upon predictions).

In the literature, the *lead time* is the interval between the date at which the prediction is made available, and the actual prediction date. While the lead time is an important parameter, the shape of its distribution law is irrelevant to the problem: either a fault is predicted at least C_p seconds in advance, and then one can checkpoint just in time before the fault, or the prediction is useless! In other words, predictions that come too late should be classified as unpredicted faults whenever they materialize as actual faults, leading to a smaller value of the predictor recall.

Fault rates

In addition to μ, the platform MTBF (see subsection 1.2.1), let μ_{P} be the mean time between predicted events (both true positive and false positive), and let μ_{NP} be the mean time between unpredicted faults (false negative). Finally, we define the mean time between events as μ_{e} (including all three event types). The relationships between μ, μ_{P}, μ_{NP}, and μ_{e} are the following:

— Rate of unpredicted faults: $\frac{1}{\mu_{\text{NP}}} = \frac{1-r}{\mu}$, since $1-r$ is the fraction of faults that are unpredicted;

— Rate of predicted faults: $\frac{r}{\mu} = \frac{p}{\mu_{\text{P}}}$, since r is the fraction of faults that are predicted, and p is the fraction of fault predictions that are correct;

— Rate of events: $\frac{1}{\mu_{\text{e}}} = \frac{1}{\mu_{\text{P}}} + \frac{1}{\mu_{\text{NP}}}$, since events are either predictions (true or false), or unpredicted faults.

Objective: waste minimization

The natural objective is to minimize the expectation of the total execution time, *makespan*, of the application. Instead, in order to ease mathematical derivations, we aim at minimizing the *waste*. The waste is the expected percentage of time lost, or "wasted", during the execution. In other words, the *waste* is the fraction of time during which the platform is not doing useful work. This definition was introduced by Wingstrom [60]. Obviously, the lower the waste, the lower the expected makespan, and reciprocally. Hence the two objectives are strongly related and minimizing one of them also minimizes the other.

4.2.2 Taking predictions into accounts

For the sake of clarity, we start with a simple algorithm (subsection 4.2.2) which we refine in subsection 4.2.2. We then compute the value of the period that minimizes the waste in subsection 4.2.2.

Simple policy

In this subsection, we consider the following algorithm:

— While no fault prediction is available, checkpoints are taken periodically with period T;

— When a fault is predicted, there are two cases: either there is the possibility to take a proactive checkpoint, or there is not enough time to do so, because we are already checkpointing (see Figures 4.1(b) and 4.1(c)). In the latter case, there is no other choice than ignoring the prediction. In the former case, we still have the possibility to ignore the prediction, but we may also decide to trust it: in fact the decision is randomly taken. With probability q, we trust the predictor and take the prediction into account (see Figures 4.1(f) and 4.1(g)), and with probability $1-q$, we ignore the prediction (see Figures 4.1(d) and 4.1(e));

— If we take the prediction into account, we take a proactive checkpoint (of length C_p) as late as possible, i.e., so that it completes right at the time when the fault is predicted to happen. After this checkpoint, we complete the execution of the period (see Figures 4.1(f) and 4.1(g));

— If we ignore the prediction, either by necessity (not enough time to take an extra checkpoint, see Figures 4.1(b) and 4.1(c)), or or by choice (with probability $1-q$, Figures 4.1(d) and 4.1(e)), we finish the current period and start a new one.

The rationale for not always trusting the predictor is to avoid taking useless checkpoints too frequently. Intuitively, the precision p of the predictor must be above a given threshold for its usage to be worthwhile. In other words, if we decide to checkpoint just before a predicted event, either we will save time by avoiding a costly re-execution if the event does correspond to an actual fault, or we will lose time by unduly performing an extra checkpoint. We need a larger proportion of the former cases, i.e., a good precision, for the predictor to be really useful. The following analysis will determine the optimal value of q as a function of the parameters C, C_p, μ, r, and p.

We could refine the approach by taking into account the amount of work already done in the current period when deciding whether to trust the predictor or not. Intuitively, the more work already done, the more important to save it, hence the more worthwhile to trust the predictor. We design such a refined strategy in subsection 4.2.2. Right now, we analyze a simpler algorithm where we decide to trust or not to trust the predictor, independently of the amount of work done so far within the period.

We analyze the algorithm in order to compute a formula for the expected waste, just as in Equation (1.13) (which we remind here):

$$\text{WASTE} = \text{WASTE}_{\text{FF}} + \text{WASTE}_{\text{fault}} - \text{WASTE}_{\text{FF}}\text{WASTE}_{\text{fault}} \tag{4.1}$$

While the value of WASTE_{FF} is unchanged ($\text{WASTE}_{\text{FF}} = \frac{C}{T}$), the value of $\text{WASTE}_{\text{fault}}$ is modified because of predictions. As illustrated in Figure 4.1, there are many different scenarios that contribute to $\text{WASTE}_{\text{fault}}$ that can be sorted into three categories:

(1) **Unpredicted faults:** This overhead occurs each time an unpredicted fault strikes, that is, on average, once every μ_{NP} seconds. Just as in Equation (1.9), the corresponding waste is $\frac{1}{\mu_{\text{NP}}}\left[\frac{T}{2} + D + R\right]$.

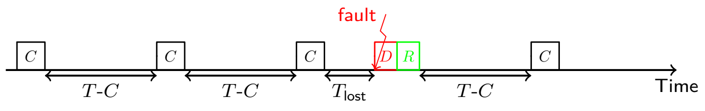
(2) **Predictions not taken into account:** The second source of waste is for predictions that are ignored. This overhead occurs in two different scenarios. First, if we do not have time to take a proactive checkpoint, we have an overhead if and only the prediction is an actual fault. This case happens with probability p. We then lose a time $t + D + R$ if the predicted fault happens a time t after the completion of the last periodic checkpoint. The expected time lost is thus

$$T^1_{\text{lost}} = \frac{1}{T}\int_0^{C_p} \left(p(t + D + R) + (1-p)0\right) dt$$

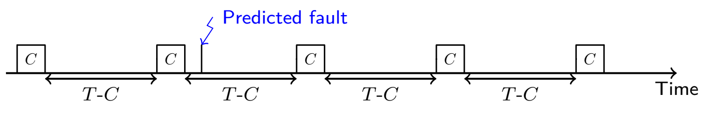
Then, if we do have time to take a proactive checkpoint but still decide to ignore the prediction, we also have an overhead if and only the prediction is an actual fault, but the expected time lost is now weighted by the probability $(1-q)$:

$$T^2_{\text{lost}} = (1-q)\frac{1}{T}\int_{C_p}^{T} \left(p(t + D + R) + (1-p)0\right) dt$$

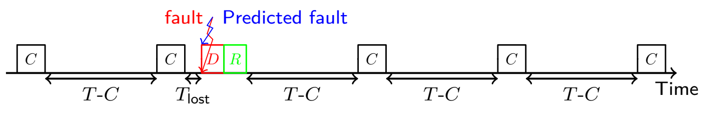
(3) **Predictions taken into account:** We now compute the overhead due to a prediction which we trust (hence we checkpoint just before its date). If the prediction is an actual fault,



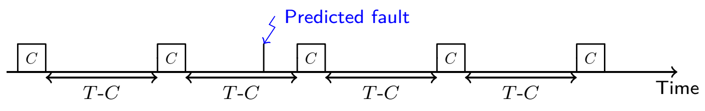
(a) Unpredicted fault



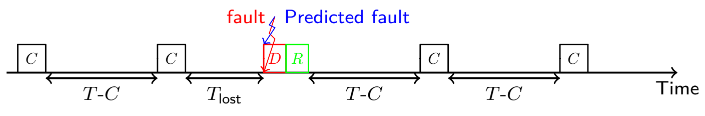
(b) Prediction cannot be taken into account - no actual fault



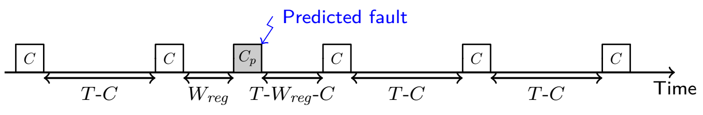
(c) Prediction cannot be taken into account - with actual fault



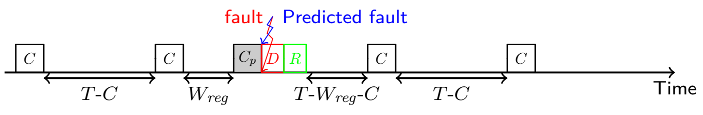
(d) Prediction not taken into account by choice - no actual fault



(e) Prediction not taken into account by choice - with actual fault



(f) Prediction taken into account - no actual fault



(g) Prediction taken into account - with actual fault

Figure 4.1: Actions taken for the different event types.

we lose $C_p + D + R$ seconds, but if it is not, we lose the unnecessary extra checkpoint time C_p. The expected time lost is now weighted by the probability q and becomes

$$T^3_{\text{lost}} = q\frac{1}{T}\int_{C_p}^{T} \left(p(C_p + D + R) + (1-p)C_p\right) dt$$

We derive the final value of $\text{W}\textsc{aste}_{\text{fault}}$:

$$\text{W}\textsc{aste}_{\text{fault}} = \frac{1}{\mu_{\text{NP}}}\left[\frac{T}{2} + D + R\right] + \frac{1}{\mu_{\text{P}}}\left[T^1_{\text{lost}} + T^2_{\text{lost}} + T^3_{\text{lost}}\right]$$

This final expression comes from the disjunction of all possibles cases, using the Law of Total Probability [59, p.23]: the waste comes either from non-predicted faults or from predictions; in the latter case, we have analyzed the three possible sub-cases and weighted them with their respective probabilities. After simplifications, we obtain

$$\text{W}\textsc{aste}_{\text{fault}} = \frac{1}{\mu}\left((1-rq)\frac{T}{2} + D + R + \frac{qr}{p}C_p - \frac{qrC_p^2}{pT}(1-p/2)\right) \tag{4.2}$$

We could now plug this expression back into Equation (4.1) to compute the value of T that minimizes the total waste. Instead, we move on to describing the refined algorithm, and we minimize the waste for the refined strategy, since it always induces a smaller waste.

Refined policy

In this subsection, we refine the approach and consider different trust strategies, depending upon the time in the period where the prediction takes place. Intuitively, the later in the period, the more likely we are inclined to trust the predictor, because the amount of work that we could lose gets larger and larger. As before, we cannot take into account a fault predicted to happen less than C_p units of time after the beginning of the period. Therefore, we focus on what happens in the period after time C_p. Formally, we now divide the interval $[C_p, T]$ into n intervals $[\beta_i; \beta_{i+1}]$ for $i \in \{0, \cdots, n-1\}$, where $\beta_0 = C_p$ and $\beta_n = T$. For each interval $[\beta_i; \beta_{i+1}]$, we trust the predictor with probability q_i. We aim at determining the values of n, β_i, and q_i that minimize the waste. As mentioned before, intuition tells us that the q_i values should be non-decreasing. We prove below a somewhat unexpected theorem: in the optimal strategy, there is either one or two different q_i values, and these values are 0 or 1. This means that we should *never* trust the predictor in the beginning of a period, and always trust it in the end of the period, without any intermediate behavior in between.

We formally express this striking result below. Let $\beta_{\text{lim}} = \frac{C_p}{p}$. The optimal strategy is provided by Theorem 8 below. We first prove the following proposition:

Proposition 9. *The values of β_i and q_i that minimize the waste satisfy the following conditions:*
(i) For all i such that $\beta_{i+1} \leq \beta_{lim}$, $q_i = 0$.
(ii) For all i such that $\beta_i \geq \beta_{lim}$, $q_i = 1$.

Proof. First we compute the waste with the refined algorithm, using Equation (4.1). The formula for $\text{W}\textsc{aste}_{\text{fault}}$ is similar to Equation (4.2) on each interval:

$$\begin{aligned}\textsc{Waste} = \frac{C}{T} + \left(1 - \frac{C}{T}\right)\Bigg[&\frac{1}{\mu_{\text{NP}}}\left(\frac{T}{2} + D + R\right)\\ &+ \frac{1}{\mu_{\text{P}}}\sum_{i=0}^{n-1}\Bigg(q_i \int_{\beta_i}^{\beta_{i+1}} \frac{(p(C_p + D + R) + (1-p)C_p)}{T}dt\\ &+ (1-q_i)\int_{\beta_i}^{\beta_{i+1}} \frac{p(t + D + R)}{T}dt\Bigg)\Bigg]\end{aligned}$$

Now, consider a fixed value of i and express the value of $\textsc{Waste}$ as a function of q_i:

$$\textsc{Waste} = K + \left(1 - \frac{C}{T}\right)\frac{q_i}{\mu_{\text{P}}}\int_{\beta_i}^{\beta_{i+1}}\left(\frac{C_p}{T} - \frac{pt}{T}\right)dt$$

where K does not depend on q_i. From the sign of the function to be integrated, one sees that $\textsc{Waste}$ is minimized when $q_i = 0$ if $\beta_{i+1} \leq \beta_{\text{lim}} = \frac{C_p}{p}$, and when $q_i = 1$ if $\beta_i \geq \beta_{\text{lim}}$. ∎

Theorem 8. *The optimal algorithm takes proactive actions if and only if the prediction falls in the interval* $[\beta_{lim}, T]$.

Proof. From Proposition 9, the values for q_i are optimally defined for every i but one: we do not know the optimal value if there exists i_0 such that $\beta_{i_0} < \beta_{\text{lim}} < \beta_{i_0+1}$. Then let us consider the waste where q_{i_0} is replaced by $q_{i_0}^{(1)}$ on $[\beta_{i_0}, \beta_{\text{lim}}]$ and by $q_{i_0}^{(2)}$ on $[\beta_{\text{lim}}, \beta_{i_0+1}]$. The new waste is necessarily smaller than the one with only q_{i_0}, since we relaxed the constraint. We know from Proposition 9 that the optimal solution is then to have $q_{i_0}^{(1)} = 0$ and $q_{i_0}^{(2)} = 1$. ∎

Let us now compute the value of the waste with the optimal algorithm. There are two cases, depending upon whether $T \leq \beta_{\text{lim}}$ or not. For values of T smaller than β_{lim}, Theorem 8 shows that the optimal algorithm never takes any proactive action; in that case the waste is given by Equation (1.14) in Chapter 1. For values of T larger than $\beta_{\text{lim}} = \frac{C_p}{p}$, we compute the waste due to predictions as

$$\begin{aligned}&\frac{1}{\mu_{\text{P}}}\frac{1}{T}\left(\int_0^{C_p/p} p(t + D + R)dt + \int_{C_p/p}^{T}(p(C_p + D + R) + (1-p)C_p)dt\right)\\ &= \frac{r}{p\mu}\left(p(D+R) + C_p - \frac{C_p^2}{2pT}\right)\end{aligned}$$

Indeed, in accordance with Theorem 8, no prediction is taken into account in the interval $[0, \frac{C_p}{p}]$, while all predictions are taken into account in the interval $[\frac{C_p}{p}, T]$. Adding the waste due to unpredicted faults, namely $\frac{1}{\mu_{\text{NP}}}\left[\frac{T}{2} + D + R\right]$, we derive

$$\textsc{Waste}_{\text{fault}} = \frac{1}{\mu}\left((1-r)\frac{T}{2} + \frac{r}{p}C_p\left(1 - \frac{1}{2p}\frac{C_p}{T}\right) + D + R\right).$$

Plugging this value into Equation (4.1), we obtain the total waste when $\frac{C_p}{p} \leq T$:

$$\begin{aligned}
\textsc{Waste} &= \frac{C}{T} + \frac{1}{\mu}\left((1-r)\frac{T}{2} + \frac{r}{p}C_p\left(1 - \frac{1}{2p}\frac{C_p}{T}\right) + D + R\right)\left(1 - \frac{C}{T}\right) \\
&= \frac{rCC_p^2}{2p^2}\frac{1}{\mu T^2} + \left(\mu C - \frac{rC_p^2}{2p^2} - C\left(\frac{rC_p}{p} + D + R\right)\right)\frac{1}{\mu T} + \frac{1-r}{2\mu}T \\
&\quad + \frac{-(1-r)\frac{C}{2} + \frac{rC_p}{p} + D + R}{\mu}
\end{aligned}$$

Altogether, the expression for the total waste becomes:

$$\begin{cases}
\textsc{Waste}_1(T) = \frac{C\left(1-\frac{D+R}{\mu}\right)}{T} + \frac{D+R-C/2}{\mu} + \frac{1}{2\mu}T & \text{if } \frac{C_p}{p} \geq T \\
\textsc{Waste}_2(T) = \frac{rCC_p^2}{2\mu p^2}\frac{1}{T^2} + \frac{\left(C\left(1-\frac{\frac{rC_p}{p}+D+R}{\mu}\right) - \frac{rC_p^2}{2\mu p^2}\right)}{T} + \frac{-(1-r)\frac{C}{2}+\frac{rC_p}{p}+D+R}{\mu} + \frac{1-r}{2\mu}T & \text{if } \frac{C_p}{p} \leq T
\end{cases} \tag{4.3}$$

One can check that when $r = 0$ (no error predicted, hence no proactive action in the algorithm), then $\textsc{Waste}_1$ and $\textsc{Waste}_2$ coincide. We also check that both values coincide for $T = \frac{C_p}{p}$. We show how to minimize the waste in Equation (4.3) in subsection 4.2.2.

Waste minimization

In this subsection we focus on minimizing the waste in Equation (4.3). Recall that, by construction, we always have to enforce the constraint $T \geq C$. First consider the case where $C \leq \frac{C_p}{p}$. On the interval $T \in [C, \frac{C_p}{p}]$, we retrieve the optimal value found in Chapter 1, and derive that $\textsc{Waste}_1$, the waste when predictions are not taken into account, is minimized for

$$T_{\textsc{NoPred}} = \max\left(C, \min\left(T_{\text{RFO}}, \frac{C_p}{p}\right)\right) \tag{4.4}$$

Indeed, the optimal value should belong to the interval $[C, \frac{C_p}{p}]$, and the function $\textsc{Waste}_1$ is convex: if the extremal solution $\sqrt{2(\mu - (D+R))C}$ does not belong to this interval, then the optimal value is one of the bounds of the interval.

On the interval $T \in \left[\frac{C_p}{p}, +\infty\right)$, we find the optimal solution by differentiating twice $\textsc{Waste}_2$ with respect to T. Writing $\textsc{Waste}_2(T) = \frac{u}{T^2} + \frac{v}{T} + w + xT$ for simplicity, we obtain $\textsc{Waste}_2''(T) = \frac{2}{T^3}\left(\frac{3u}{T} + v\right)$. Here, a key parameter is the sign of :

$$v = \left(C\left(1 - \frac{\frac{rC_p}{p} + D + R}{\mu}\right) - \frac{rC_p^2}{2\mu p^2}\right)$$

We detail the case $v \geq 0$ in the following, because it is the most frequent with realistic parameter sets; we do have $v \geq 0$ for all the whole range of simulations in subsection 4.2.3. For the sake of completeness, we will briefly discuss the case $v < 0$ in the comments below.

When $v \geq 0$, we have $\textsc{Waste}_2''(T) \geq 0$, so that $\textsc{Waste}_2$ is convex on the interval $\left[\frac{C_p}{p}, +\infty\right)$ and admits a unique minimum T_{extr}. Note that T_{extr} can be computed either numerically or using Cardano's method, since it is the unique real root of a polynomial of degree 3. The optimal solution on $\left[\frac{C_p}{p}, +\infty\right)$ is then: $T_{\text{Pred}} = \max\left(T_{\text{extr}}, \frac{C_p}{p}\right)$.

It remains to consider the case where $\frac{C_p}{p} < C$. In fact, it suffices to add the constraint that the value of T_{Pred} should be greater than C, that is:

$$T_{\text{Pred}} = \max\left(C, \max\left(T_{\text{extr}}, \frac{C_p}{p}\right)\right) \tag{4.5}$$

Finally, the optimal solution for the waste is given by the minimum of the following two values:

$$\begin{cases} \frac{C\left(1-\frac{D+R}{\mu}\right)}{T_{\text{NoPred}}} + \frac{D+R-C/2}{\mu} + \frac{1}{2\mu}T_{\text{NoPred}} \\ \frac{rCC_p^2}{2\mu p^2}\frac{1}{T_{\text{Pred}}^2} + \frac{\left(C\left(1-\frac{\frac{rC_p}{p}+D+R}{\mu}\right)-\frac{rC_p^2}{2\mu p^2}\right)}{T_{\text{Pred}}} + \frac{-(1-r)\frac{C}{2}+\frac{rC_p}{p}+D+R}{\mu} + \frac{1-r}{2\mu}T_{\text{Pred}} \end{cases}$$

We make a few observations:

— Just as for Equation (1.15) in Chapter 1, mathematical rigor calls for capping the values of D, R, C, C_p and T in front of the MTBF. The only difference is that we should replace μ by μ_e: this is to account for the occurrence rate of all events, be they unpredicted faults or predictions.

— While the expression of the waste looks complicated, the numerical value of the optimal period can easily be computed in all cases. We have dealt with the case $v \geq 0$, where v is the coefficient of $1/T$ in $\text{Waste}_2(T) = \frac{u}{T^2} + \frac{v}{T} + w + xT$. When $v < 0$ we only needs to compute all the nonnegative real roots of a polynomial of degree 3, and check which one leads to the best value. More precisely, these root(s) partition the admissible interval $\left[\frac{C_p}{p}, +\infty\right)$ into several sub-intervals, and the optimal value is either a root or a sub-interval bound.

— In many practical situations, when μ is large enough, we can dramatically simplify the expression of $\text{Waste}_2(T)$: we have $T = O(\sqrt{\mu})$, the term $\frac{u}{T^2}$ becomes negligible, checkpoint parameters become negligible in front of μ, and we derive the approximated value $\sqrt{\frac{2\mu C}{1-r}}$. This value can be seen as an extension of Equation (1.15) giving T_{RFO}, where μ is replaced by $\frac{\mu}{1-r}$: faults are replaced by non-predicted faults, and the overhead due to false predictions is negligible. As a word of caution, recall that this conclusion is valid only when μ is very large in front of all other parameters.

4.2.3 Simulation results

We start by presenting the simulation framework (subsection 4.2.3). Then we report results using synthetic traces (subsection 4.2.3) and log-based traces (subsection 4.2.3). Finally, we assess the respective impact of the two key parameters of a predictor, its recall and its precision, on checkpointing strategies (subsection 4.2.3).

Simulation framework

Scenario generation – In order to check the accuracy of our model and of our analysis, and to assess the potential benefits of predictors, we study the performance of our new solutions and of pre-existing ones using a discrete-event simulator. The simulation engine generates a random trace of faults. Given a set of p processors, a failure trace is a set of failure dates for each processor over a fixed time horizon h (set to 2 years). Given the distribution of inter-arrival times at a processor, for each processor we generate a trace via independent sampling until the target time horizon is reached. The job start time is assumed to be one-year to avoid side-effects related to the synchronous initialization of all nodes/processors. We consider two types of failure traces, namely synthetic and log-based.

Synthetic failure traces – The simulation engine generates a random trace of faults parameterized either by an Exponential fault distribution or by Weibull distribution laws with shape parameter either 0.5 or 0.7. Note that Exponential faults are widely used for theoretical studies, while Weibull faults are representative of the behavior of real-world platforms [64, 63, 65, 66]. For example, Heien et al. [66] have studied the failure distribution for 6 sources of failures (storage devices, NFS, batch system, memory and processor cache errors, etc.), and the aggregate failure distribution. They have shown that the aggregate failure distribution is best modeled by a Weibull distribution with a shape parameter that is between 0.5841 and 0.7097.

The Jaguar platform, which comprised $N = 45,208$ processors, is reported to have experienced about one fault per day [46], which leads to an individual (processor) MTBF μ_{ind} equal to $\frac{45{,}208}{365} \approx 125$ years. Therefore, we set the individual (processor) MTBF to $\mu_{\text{ind}} = 125$ years. We let the total number of processors N vary from $N = 16,384$ to $N = 524,288$, so that the platform MTBF μ varies from $\mu = 4,010$ min (about 2.8 days) down to $\mu = 125$ min (about 2 hours). Whatever the underlying failure distribution, it is scaled so that its expectation corresponds to the platform MTBF μ. The application size is set to $\textsc{Time}_{\text{base}} = 10,000$ years/N.

Log-based failure traces – To corroborate the results obtained with synthetic failure traces, and to further assess the performance of our algorithms, we also perform simulations using the failure logs of two production clusters. We use logs of the largest clusters among the preprocessed logs in the *Failure trace archive* [67], i.e., for clusters at the Los Alamos National Laboratory [63]. In these logs, each failure is tagged by the node —and not the processor— on which the failure occurred. Among the 26 possible clusters, we opted for the logs of the only two clusters with more than 1,000 nodes. The motivation is that we need a sample history sufficiently large to simulate platforms with more than ten thousand nodes. The two chosen logs are for clusters 18 (LANL18) and 19 (LANL19) in the archive (referred to as 7 and 8 in [63]). For each log, we record the set $\mathcal{S}$ of availability intervals. The discrete failure distribution for the simulation is generated as follows: the conditional probability $\mathbb{P}(X \geq t \mid X \geq \tau)$ that a node stays up for a duration t, knowing that it has been up for a duration τ, is set to the ratio of the number of availability durations in $\mathcal{S}$ greater than or equal to t, over the number of availability durations in $\mathcal{S}$ greater than or equal to τ.

The two clusters used for computing our log-based failure distributions consist of 4-processor nodes. Hence, to simulate a platform of, say, 2^{16} processors, we generate 2^{14} failure traces, one for each 4-processor node. In the logs the individual (processor) MTBF is $\mu_{\text{ind}} = 691$ days for the LANL18 cluster, and $\mu_{\text{ind}} = 679$ days for the LANL19 cluster. The LANL18 and LANL19 traces are logs for systems which comprised 4,096 processors. Using these logs to generate traces for a system made of $524,288$ processors, as the largest platforms we consider with synthetic failure traces, would lead to an obvious risk of oversampling. Therefore, we limit the size of the log-based traces we generate: we let the total number of processors N varies from $N = 1,024$ to $N = 131,072$, so that the platform MTBF μ varies from $\mu = 971$ min (about 16 hours) down to $\mu = 7.5$ min. The application size is set to $\textsc{Time}_{\text{base}} = 250$ years/N.

Predicted failures and false predictions – Once we have generated a failure trace, we need to determine which faults are predicted and which are not. In order to do so, we consider all faults in a trace one by one. For each of them, we randomly decide, with probability r, whether it is predicted.

We use the simulation engine to generate a random trace of false predictions. The main problem is to decide the shape of the distribution that false predictions should follow. To the best of our knowledge, no published study ever addressed that problem. For synthetic failure traces, we report results when false predictions follow the same distribution than faults (except, of course, that both distributions do not have the same mean value). Results are quite similar when false predictions are generated according to a uniform distribution. For log-based failures, we only report results when false predictions are generated according to a uniform distribution (because we believe that scaling down a discrete, actual distribution may not be meaningful).

The distribution of false predictions is always scaled so that its expectation is equal to $\frac{\mu_P}{1-p} = \frac{p\mu}{r(1-p)}$, the inter-arrival time of false predictions. Finally, the failure trace and the false-prediction trace are merged to produce the final trace including all events (true predictions, false predictions, and non predicted faults). Each reported value is the average over 100 randomly generated instances.

Checkpointing, recovery, and downtime costs – The experiments use parameters that are representative of current and forthcoming large-scale platforms [68, 69]. We take $C = R = 10$ min, and $D = 1$ min for the synthetic failure traces. For the log-based traces we consider smaller platforms. Therefore, we take $C = R = 1$ min, and $D = 6$s. Whatever the trace, we consider three scenarios for the proactive checkpoints: either proactive checkpoints are (i) exactly as expensive as periodic ones ($C_p = C$), (ii) ten times cheaper ($C_p = 0.1C$), and (iii) two times more expensive ($C_p = 2C$).

Heuristics – In the simulations, we compare four checkpointing strategies:

— RFO is the checkpointing strategy of period $T = \sqrt{2(\mu - (D + R))C}$ (see Chapter 1).
— OptimalPrediction is the refined algorithm described in subsection 4.2.2.
— To assess the quality of each strategy, we compare it with its BestPeriod counterpart, defined as the same strategy but using the best possible period T. This latter period is computed via a brute-force numerical search for the optimal period (each tested period is evaluated on 100 randomly generated traces, and the period achieving the best average performance is elected as the “best period”).

Fault predictors – We experiment using the characteristics of two predictors from the literature: one accurate predictor with high recall and precision [51], namely with $p = 0.82$ and $r = 0.85$, and another predictor with intermediate recall and precision [52], namely with $p = 0.4$ and $r = 0.7$.

In practice, a predictor will not be able to predict the exact time at which a predicted fault will strike the system. Therefore, in the simulations, when a predictor predicts that a failure will strike the system at a date t (true prediction), the failure actually occurs exactly at time t for heuristic OptimalPrediction, and between time t and time $t + 2C$ for heuristic InexactPrediction (the probability of fault is uniformly distributed in the time-interval). OptimalPrediction can thus be seen as a best case. The comparison between Optimal-Prediction and InexactPrediction enables us to assess the impact of the time imprecision of predictions, and to show that the obtained results are quite robust to this type of imprecision. The choice of an interval length of $2C$ is quite arbitrary. For synthetic traces, this corresponds to 1,200 s, which is quite a significant imprecision.

Simulations with synthetic traces

Figures 4.2 and 4.3 show the average waste degradation for the two checkpointing policies, and for their BestPeriod counterparts, for both predictors. The waste is reported as a function of the number of processors N. We draw the plots as a function of the number of processors N rather than of the platform MTBF $\mu = \mu_{\text{ind}}/N$, because it is more natural to see the waste increase with larger platforms. However, recall that this work is agnostic of the granularity of the processing elements and intrinsically focuses on the impact of the MTBF on the waste.

We also report job execution times, in Table 4.2 when fault distribution follows an Exponential distribution law, and in Tables 4.3 and 4.4 for a Weibull distribution law with shape parameter $k = 0.7$ and $k = 0.5$ respectively.

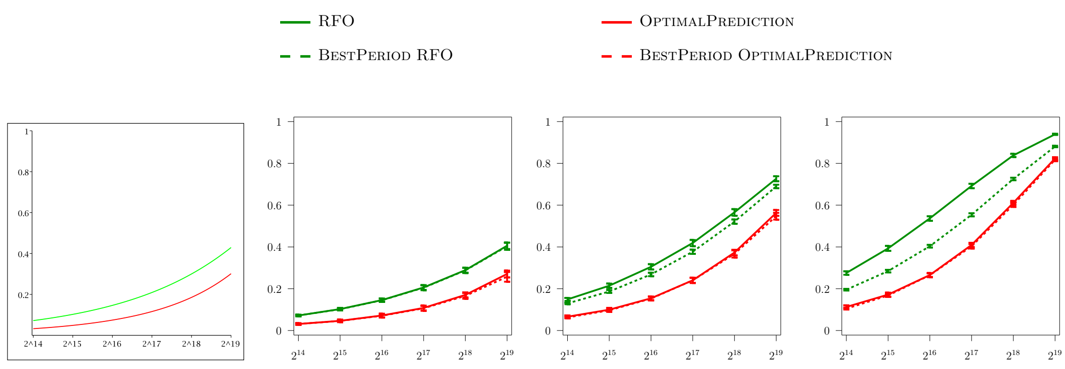
Validation of the theoretical study – We used Maple to analytically compute and plot the optimal value of the waste for both the algorithm taking predictions into account, OptimalPrediction, and for the algorithm ignoring them, RFO. In order to check the accuracy of our model, we have compared these results with results obtained with the discrete-event simulator.

We first observe that there is a very good correspondence between analytical results and simulations in Figures 4.2 and 4.3. In particular, the Maple plots and the simulations for Exponentially distributed faults are very similar. This shows the validity of the model and of its analysis. Another striking result is that OptimalPrediction has the same waste as its BestPeriod counterpart, even for Weibull fault distributions, in all but the most extreme cases. In the other cases, the waste achieved by OptimalPrediction is very close to that of its BestPeriod counterpart. This demonstrates the very good quality of our checkpointing period T_{Pred}. These conclusions are valid regardless of the cost ratio of periodic and proactive checkpoints.

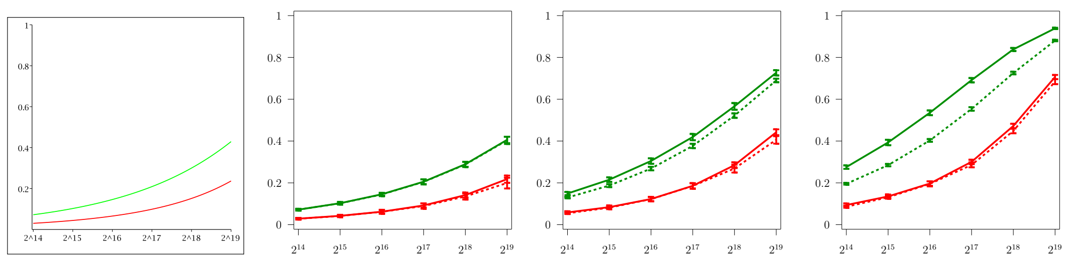
In Tables 4.2 through 4.4 we report the execution times obtained when using the expression of T given by Young [53] and Daly [54] (denoted respectively as Young and Daly) to assess whether T_{RFO} is a better approximation. (Recall that these three approaches ignore the predictions, which explains why the numbers are identical on both sides of each table.) The expressions of T given by Young, Daly, and RFO are identical for Exponential distributions and the three heuristics achieve the same performance (Table 4.2). This confirms the analytical evaluation of Table 1.2 in Chapter 1. For Weibull distributions (Tables 4.3 and 4.4), RFO achieves lower makespan, and the difference becomes even more significant as the size of the platform increases. Moreover, it is striking to observe in Table 4.4 that job execution time increases together with the number for processors (from $N = 2^{16}$ to $N = 2^{19}$) if the checkpointing period is Daly or Young. On the contrary, job execution time (rightfully) decreases when using RFO, even if the decrease is moderate with respect to the increase of the platform size. Altogether, the main (striking) conclusion is that RFO should be preferred to both classical approaches for Weibull distributions.

The benefits of prediction – The second observation is that the prediction is useful for the vast majority of the set of parameters under study! In addition, when proactive checkpoints are cheaper than periodic ones, the benefits of fault prediction are increased. On the contrary, when proactive checkpoints are more expensive than periodic ones, the benefits of fault prediction are greatly reduced. One can even observe that the waste with prediction is not better than without prediction in the following scenario: $C_p = 2C$, and using the limited-quality predictor ($p = 0.4$, $r = 0.7$) with 2^{19} processors, see Figures 4.3(i),(j),(k), and (l).

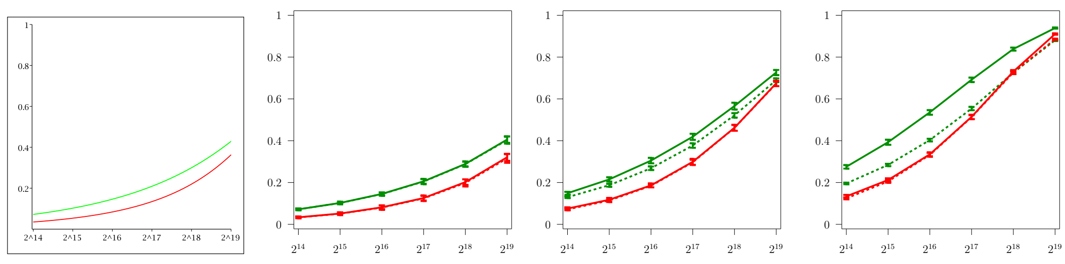
In Tables 4.2 through 4.4 we compute the gain (expressed in percentage) achieved by OptimalPrediction over RFO. As a general trend, we observe that the gains due to predictions



(a) Maple (b) Exponential (c) Weibull $k = 0.7$ (d) Weibull $k = 0.5$

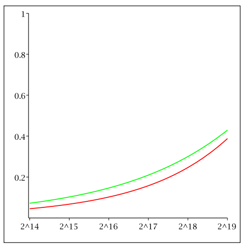


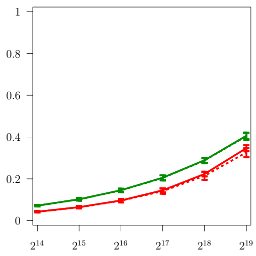
(e) Maple (f) Exponential (g) Weibull $k = 0.7$ (h) Weibull $k = 0.5$

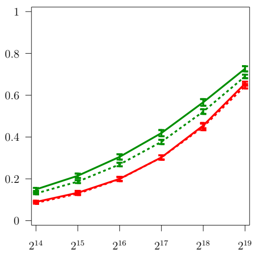


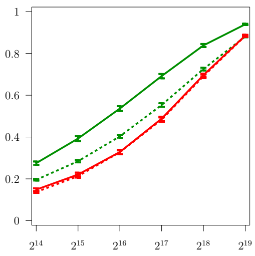
(i) Maple (j) Exponential (k) Weibull $k = 0.7$ (l) Weibull $k = 0.5$

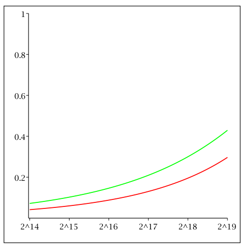
Figure 4.2: Waste (y-axis) for the different heuristics as a function of the platform size (x-axis), with $p = 0.82$, $r = 0.85$, $C_p = C$ (first row), $C_p = 0.1C$ (second row), or $C_p = 2C$ (third row) and with a trace of false predictions parametrized by a distribution identical to the distribution of the failure trace.

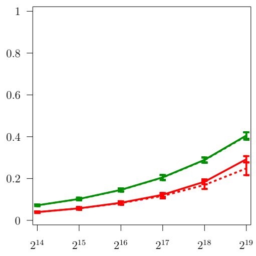

(a) Maple

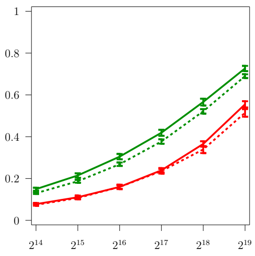

(b) Exponential

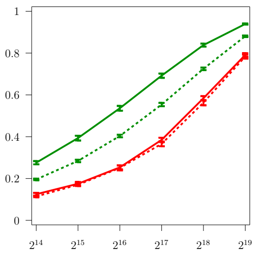

(c) Weibull $k = 0.7$

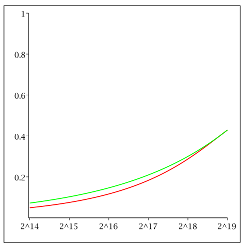

(d) Weibull $k = 0.5$

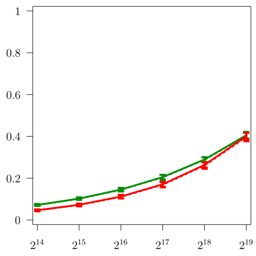

(e) Maple

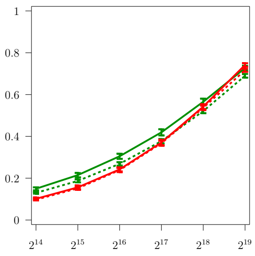

(f) Exponential


(g) Weibull $k = 0.7$


(h) Weibull $k = 0.5$


(i) Maple


(j) Exponential


(k) Weibull $k = 0.7$

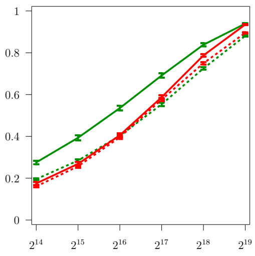

(l) Weibull $k = 0.5$

Figure 4.3: Waste (y-axis) for the different heuristics as a function of the platform size (x-axis), with $p = 0.4$, $r = 0.7$, $C_p = C$ (first row), $C_p = 0.1C$ (second row), or $C_p = 2C$ (third row) and with a trace of false predictions parametrized by a distribution identical to the distribution of the failure trace.

are more important when the distribution law is further apart from an Exponential distribution. Indeed, the largest gains are when the fault distribution follows a Weibull law of parameter 0.5. Using OptimalPrediction in conjunction with a “good” fault predictor we report gains up to 66% when there is a large number of processors (2^{19}). The gain is still of 37% with 2^{16} processors. Using a predictor with limited recall and precision, OptimalPrediction can still decrease the execution time by 47% with 2^{19} processors, and 31% with 2^{16} processors. In all tested cases, the decrease of the execution times is significant. Gains are less important with Weibull laws of shape parameter $k = 0.7$, however they are still reaching a minimum of 13% with 2^{16} processors, and up to 38% with 2^{19} processors. Finally, gains are further reduced with an Exponential law. They are still reaching at least 5% with 2^{16} processors, and up to 19% with 2^{19} processors.

The performance of InexactPrediction shows that using a fault predictor remains largely beneficial even in the presence of large uncertainties on the time the predicted faults will actually occur (see Tables 4.2, 4.3, and 4.4). When $N = 2^{16}$ the degradation with respect to OptimalPrediction is of 3% for a Weibull law with shape parameter $k = 0.7$, and the minimum gain over RFO is still of 10%. When the shape parameter of the Weibull law is $k = 0.5$, the degradation is of 7% when, for a minimum gain of 26% over RFO.

$C_p = C$	Execution time (in days) ($p = 0.82$, $r = 0.85$)		Execution time (in days) ($p = 0.4$, $r = 0.7$)	
	2^{16} procs	2^{19} procs	2^{16} procs	2^{19} procs
Young	65.2	11.7	65.2	11.7
Daly	65.2	11.8	65.2	11.8
RFO	65.2	11.7	65.2	11.7
OptimalPrediction	60.0 (8%)	9.5 (19%)	61.7 (5%)	10.7 (8%)
InexactPrediction	60.6 (7%)	10.2 (13%)	62.3 (4%)	11.4 (3%)

Table 4.2: Job execution times for an Exponential distribution, and gains due to the fault predictor (with respect to the performance of RFO).

$C_p = C$	Execution time (in days) ($p = 0.82$, $r = 0.85$)		Execution time (in days) ($p = 0.4$, $r = 0.7$)	
	2^{16} procs	2^{19} procs	2^{16} procs	2^{19} procs
Young	81.3	30.1	81.3	30.1
Daly	81.4	31.0	81.4	31.0
RFO	80.3	25.5	80.3	25.5
OptimalPrediction	65.9 (18%)	15.9 (38%)	69.7 (13%)	20.2 (21%)
InexactPrediction	68.0 (15%)	20.3 (20%)	72.0 (10%)	24.6 (4%)

Table 4.3: Job execution times for a Weibull distribution with shape parameter $k = 0.7$, and gains due to the fault predictor (with respect to the performance of RFO).

Simulations with log-based traces

Figure 4.4 shows the average waste degradation for the two checkpointing policies, and for their BestPeriod counterparts, for both predictors, both traces, and the three scenarios for

$C_p = C$	Execution time (in days) ($p = 0.82$, $r = 0.85$)		Execution time (in days) ($p = 0.4$, $r = 0.7$)	
	2^{16} procs	2^{19} procs	2^{16} procs	2^{19} procs
Young	125.5	171.8	125.5	171.8
Daly	125.8	184.7	125.8	184.7
RFO	120.2	114.8	120.2	114.8
OptimalPrediction	75.9 (37%)	39.5 (66%)	83.0 (31%)	60.8 (47%)
InexactPrediction	82.0 (32%)	60.8 (47%)	89.4 (26%)	76.6 (33%)

Table 4.4: Job execution times for a Weibull distribution with shape parameter $k = 0.5$, and gains due to the fault predictor (with respect to the performance of RFO).

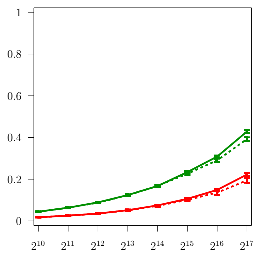
proactive checkpoints. Tables 4.5 and 4.6 present job execution times for RFO, OptimalPrediction, and InexactPrediction, for both traces and for platform sizes smaller than as the ones reported in Tables 4.2 through 4.4 for synthetic traces. The waste for RFO is closer to its BestPeriod counterpart with log-based traces than with Weibull-based traces. As a consequence, when prediction with OptimalPrediction is beneficial, it is beneficial with respect to both RFO, and to RFO's BestPeriod.

Overall, we observe similar results and reach the same conclusions with log-based traces as with synthetic ones. The waste of OptimalPrediction is very close to that of its BestPeriod counterpart for platforms containing up to 2^{16} processors. This demonstrates the validity of our analysis for the actual traces considered. The waste of OptimalPrediction is often significantly larger than that of its BestPeriod counterpart for platforms containing 2^{17} processors. The problem with the largest considered platforms may be due to oversampling. Indeed, the original logs recorded events for platforms comprising only 4,096 processors and respectively contained only 3,010 and 2,343 availability intervals.

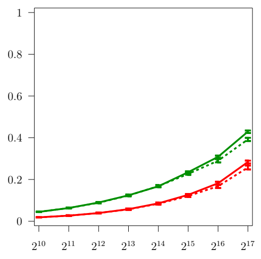
As with synthetic failure traces, prediction turns out to be useful for the vast majority of tested configurations. The only cases when prediction is not useful is with the "bad" predictor ($r = 0.7$ and $p = 0.4$), when the cost of proactive checkpoint is larger than the cost of periodic checkpoints ($C_p = 2C$), and when considering the largest of platforms ($N = 2^{17}$). This extreme case is, however, the only one for which prediction is not beneficial. It is not surprising that predictions are not useful when there are a lot of false predictions that require the use of expensive proactive actions. Looking at Tables 4.5 and 4.6, one could remark that performance gains due to the predictions are similar to the ones observed with Exponential-based traces, and are significantly smaller than the ones observed with Weibull-based traces. However, recall that we remarked that gains increase with the size of the platform, and that we consider smaller platforms when using log-based traces.

Finally, the imprecision related to the time where predicted faults strike, induces a performance degradation. However, this degradation is rather limited for the most efficient of the two predictors considered, or when the platform size is not too large.

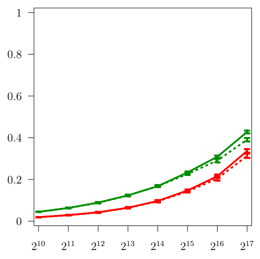
RFO
OptimalPrediction
BestPeriod RFO
BestPeriod OptimalPrediction



(a) $C_p = 0.1C$

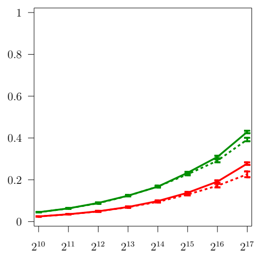


(b) $C_p = 1C$

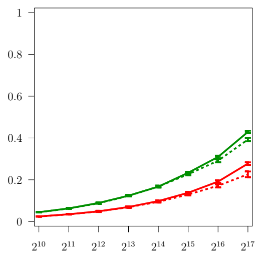


(c) $C_p = 2C$

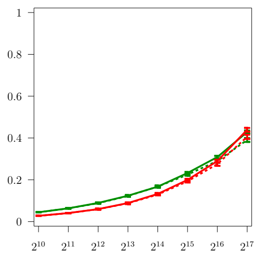
LANL18 cluster with $p = 0.82$, $r = 0.85$.



(d) $C_p = 0.1C$

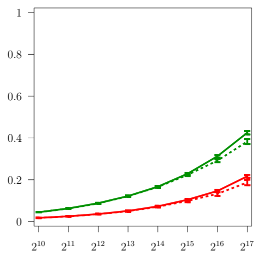


(e) $C_p = 1C$

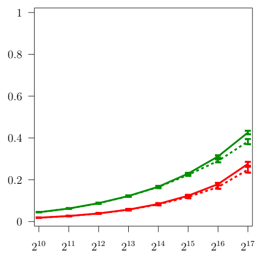


(f) $C_p = 2C$

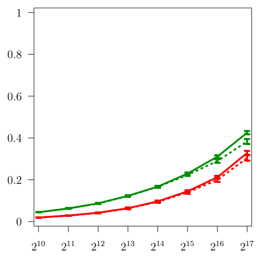
LANL18 cluster with $p = 0.4$, $r = 0.7$.



(g) $C_p = 0.1C$

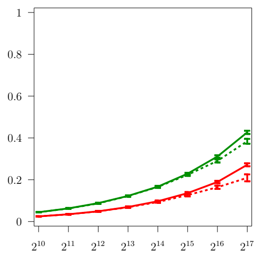


(h) $C_p = 1C$

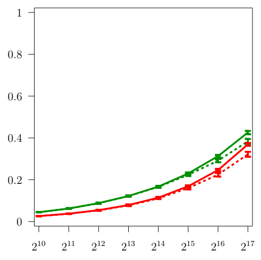


(i) $C_p = 2C$

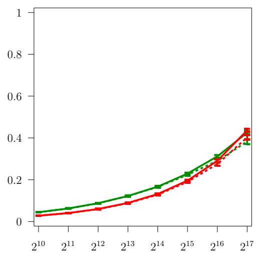
LANL19 cluster with $p = 0.82$, $r = 0.85$.



(j) $C_p = 0.1C$



(k) $C_p = 1C$



(l) $C_p = 2C$

LANL19 cluster with $p = 0.4$, $r = 0.7$.

Figure 4.4: Waste (y-axis) for the different heuristics as a function of the platform size (x-axis) with failures based on the failure log of LANL clusters 18 and 19.

$C_p = C$	Execution time (in days) $(p = 0.82,\ r = 0.85)$		Execution time (in days) $(p = 0.4,\ r = 0.7)$	
	2^{14} procs	2^{17} procs	2^{14} procs	2^{17} procs
RFO	26.8	4.88	26.8	4.88
OptimalPrediction	24.4 (9%)	3.89 (20%)	25.2 (6%)	4.44 (9%)
InexactPrediction	24.7 (8%)	4.20 (14%)	25.5 (5%)	4.73 (3%)

Table 4.5: Job execution times with failures based on the failure log of LANL18 cluster, and gains due to the fault predictor (with respect to the performance of RFO).

$C_p = C$	Execution time (in days) $(p = 0.82,\ r = 0.85)$		Execution time (in days) $(p = 0.4,\ r = 0.7)$	
	2^{14} procs	2^{17} procs	2^{14} procs	2^{17} procs
RFO	26.8	4.86	26.8	4.86
OptimalPrediction	24.4 (9%)	3.85 (21%)	25.2 (6%)	4.42 (9%)
InexactPrediction	24.6 (8%)	4.14 (15%)	25.4 (5%)	4.71 (3%)

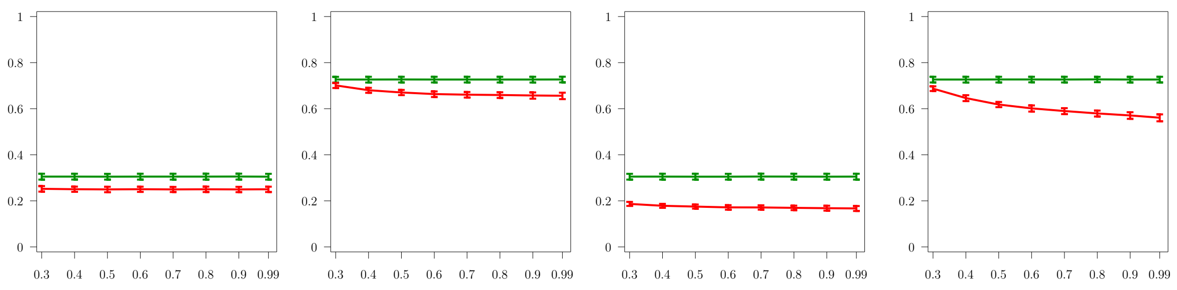
Table 4.6: Job execution times with failures based on the failure log of LANL19 cluster, and gains due to the fault predictor (with respect to the performance of RFO).

Recall vs. precision

In this subsection, we assess the impact of the two key parameters of the predictor, its recall r and its precision p. To this purpose, we conduct simulations with synthetic traces, where one parameter is fixed while the other varies. We choose two platforms, a smaller one with $N = 2^{16}$ processors (or a MTBF $\mu = 1,000\ min$) and a larger one with $N = 2^{19}$ processors (or a MTBF $\mu = 125\ min$). In both cases we study the impact of the predictor characteristics assuming a Weibull fault distribution with shape parameter either 0.5 or 0.7, under the scenario $C_p = C$.

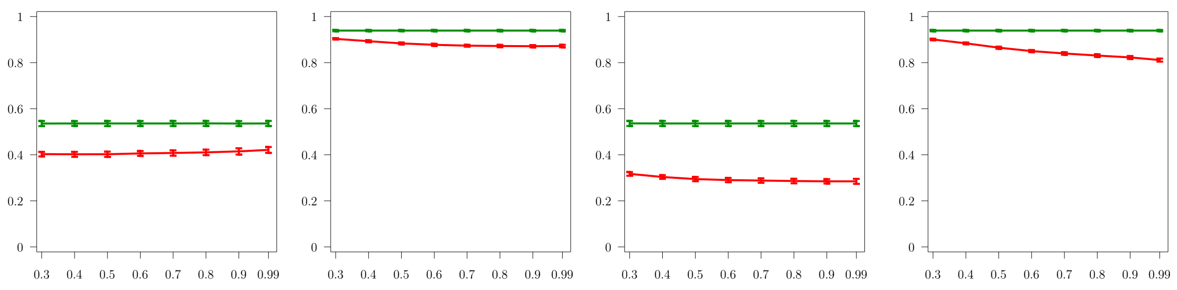
In Figures 4.5 and 4.6, we fix the value of r (either $r = 0.4$ or $r = 0.8$) and we let p vary from 0.3 to 0.99. In the four plots, we observe that the precision has a minor impact on the waste, whether it is with a Weibull distribution of shape parameter 0.7 (Figure 4.5), or a Weibull distribution of shape parameter 0.5 (Figure 4.6). In Figures 4.7 and 4.8, we conduct the converse experiment and fix the value of p (either $p = 0.4$ or $p = 0.8$), letting r vary from 0.3 to 0.99. Here we observe that increasing the recall significantly improves performance, in all but one configuration. In the configuration where improving the recall does not make a (significant) difference, there is a very large number of faults and a low precision, hence a large number of false predictions which negatively impact the performance whatever the value of the recall.

Altogether we conclude that it is more important (for the design of future predictors) to focus on improving the recall r rather than the precision p, and our results can help quantify this statement. We provide an intuitive explanation as follows: unpredicted faults prove very harmful and heavily increase the waste, while unduly checkpointing due to false predictions (usually) turns out to induce a smaller overhead.



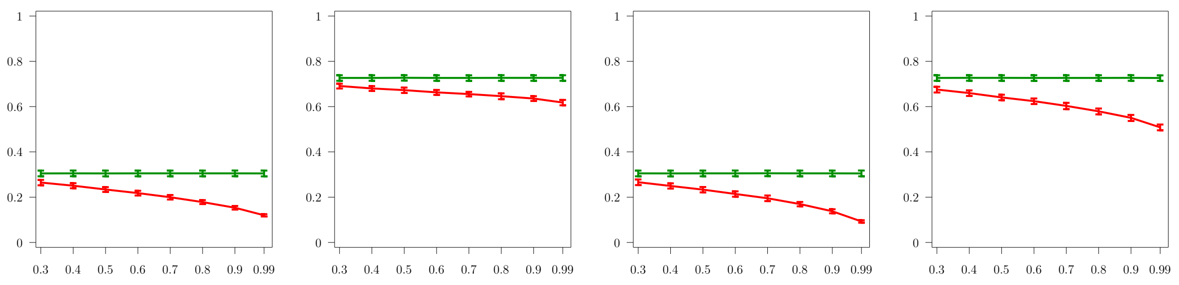
(a) $r = 0.4,\ N = 2^{16}$ (b) $r = 0.4,\ N = 2^{19}$ (c) $r = 0.8,\ N = 2^{16}$ (d) $r = 0.8,\ N = 2^{19}$

Figure 4.5: Waste (y-axis) as a function of the precision (x-axis) for a fixed recall ($r = 0.4$ and $r = 0.8$) and for a Weibull distribution of faults (with shape parameter $k = 0.7$).



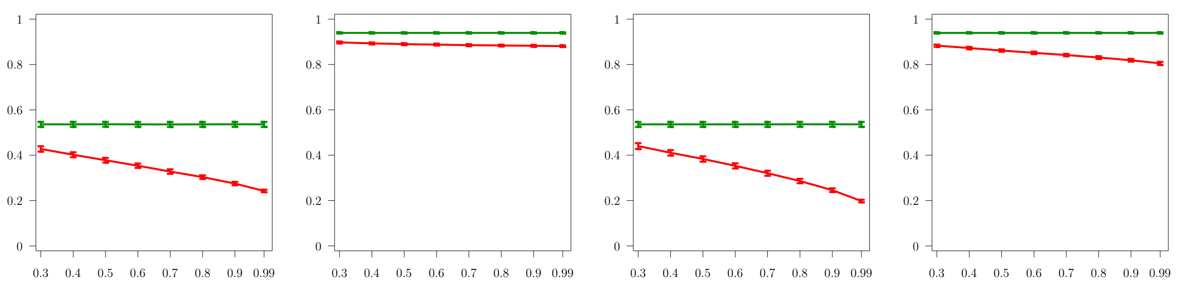
(a) $r = 0.4,\ N = 2^{16}$ (b) $r = 0.4,\ N = 2^{19}$ (c) $r = 0.8,\ N = 2^{16}$ (d) $r = 0.8,\ N = 2^{19}$

Figure 4.6: Waste (y-axis) as a function of the precision (x-axis) for a fixed recall ($r = 0.4$ and $r = 0.8$) and for a Weibull distribution of faults (with shape parameter $k = 0.5$).



(a) $p = 0.4,\ N = 2^{16}$ (b) $p = 0.4,\ N = 2^{19}$ (c) $p = 0.8,\ N = 2^{16}$ (d) $p = 0.8,\ N = 2^{19}$

Figure 4.7: Waste (y-axis) as a function of the recall (x-axis) for a fixed precision ($p = 0.4$ and $p = 0.8$) and for a Weibull distribution (k=0.7).



(a) $p = 0.4,\ N = 2^{16}$ (b) $p = 0.4,\ N = 2^{19}$ (c) $p = 0.8,\ N = 2^{16}$ (d) $p = 0.8,\ N = 2^{19}$

Figure 4.8: Waste (y-axis) as a function of the recall (x-axis) for a fixed precision ($p = 0.4$ and $p = 0.8$) and for a Weibull distribution (k=0.5).

4.3 Predictor with a prediction window

In this section, we refine the work on the impact of fault prediction techniques on coordinated checkpointing strategies.

Assume now that some fault prediction system is available. We remind that such a system is characterized by two critical parameters, its recall r, which is the fraction of faults that are indeed predicted, and its precision p, which is the fraction of predictions that are correct (i.e., correspond to actual faults). In the simple case where predictions are exact-date predictions, Gainaru et al. [49] and our previous work (Section 4.2) have independently shown that the optimal checkpointing period becomes $T_{\text{opt}} = \sqrt{\frac{2\mu C}{1-r}}$. This latter expression is valid only when μ is large enough. This expression can be seen as an extension of Young's formula where μ is replaced by $\frac{\mu}{1-r}$: faults are replaced by non-predicted faults, and the overhead due to false predictions is negligible. A more accurate expression for the optimal checkpointing period is available in Chapter 1.

This Section deals with the realistic case (see [51, 50] and related work in section 1.1) where the predictor system does not provide exact dates for predicted events, but instead provides *prediction windows*. A *prediction window* is a time interval of length I during which the predicted event is likely to happen. Intuitively, one is more at risk during such an interval than in the absence of any prediction, hence the need to checkpoint more frequently. But with which period? Should we take into account all predictions? And what is the size of the prediction window above which it proves worthwhile to use a different (smaller) checkpointing period during the prediction windows? It turns out that the answer to those questions is dramatically more complicated than when using exact-date predictions.

This Section is organized as follows. First we detail the framework in subsection 4.3.1. In subsection 4.3.2 we describe the new checkpointing policies with prediction windows, and show how to compute the optimal checkpointing periods that minimize the platform waste. subsection 4.3.3 is devoted to simulations. Finally, we present concluding remarks in subsection 4.4.

4.3.1 Framework

This Section uses a very similar model to the one introduced in subsection 4.2.1. The only difference is the definition of the fault predictor.

Fault predictor

A fault predictor is a mechanism that is able to predict that some faults will take place, *within some time-interval window*. Again, we assume that the predictor is able to generate its predictions early enough so that a *proactive* checkpoint can indeed be taken before or during the event. A first proactive checkpoint will typically be taken just before the beginning of the prediction window, and possibly several other ones will be taken inside the prediction window, if its size I is large enough.

As in Section 4.2, the accuracy of the fault predictor is characterized by two quantities, the *recall* and the *precision*. The recall r is the fraction of faults that are predicted while the precision p is the fraction of fault predictions that are correct.

4.3.2 Checkpointing strategies

In this subsection, we introduce the new checkpointing strategies, and we determine the waste that they induce. We then proceed to computing the optimal period for each strategy.

Description of the different strategies

We consider the following general scheme:

1. While no fault prediction is available, checkpoints are taken periodically with period T;
2. When a fault is predicted, we decide whether to take the prediction into account or not. This decision is randomly taken: with probability q, we trust the predictor and take the prediction into account, and, with probability $1 - q$, we ignore the prediction;
3. If we decide to trust the predictor, we use various strategies, depending upon the length I of the prediction window.

Before describing the different strategies in situation (3), we point out that the rationale for not always trusting the predictor is to avoid taking useless checkpoints too frequently. Indeed, the precision p of the predictor must be above a given threshold for its usage to be worthwhile. In other words, if we decide to checkpoint just before a predicted event, either we will save time by avoiding a costly re-execution if the event does correspond to an actual fault, or we will lose time by unduly performing an extra checkpoint.

Now, to describe the strategies used when we trust a prediction (situation (3)), we define two *modes* for the scheduling algorithm. The **Regular** mode is used when no fault prediction is available, or when a prediction is available but we decide to ignore it (with probability $1-q$). In regular mode, we use periodic checkpointing with period T_R. Intuitively, T_R corresponds to the checkpointing period T of Section 4.2. The **Proactive** mode is used when a fault prediction is available and we decide to trust it, a decision taken with probability q. Consider such a trusted prediction made for a prediction window $[t_0, t_0 + I]$. Several strategies can be envisioned:
(1) Instant, for *Instantaneous*– The first strategy (see Figure 4.9) is to ignore the time-window and to execute the same algorithm as if the predictor had given an exact date prediction at time t_0. The algorithm interrupts the current period (of scheduled length T_R), checkpoints during the interval $[t_0 - C_p, t_0]$, and then returns to regular mode: at time t_0, it resumes the work needed to complete the interrupted period of the regular mode.

(2) NoCkptI, for *No checkpoint during prediction window*– The second strategy (see Figure 4.10) is intended for a short prediction window: instead of ignoring it, we acknowledge it, but make the decision not to checkpoint during it. As in the first strategy, the algorithm interrupts the current period (of scheduled length T_R), and checkpoints during the interval

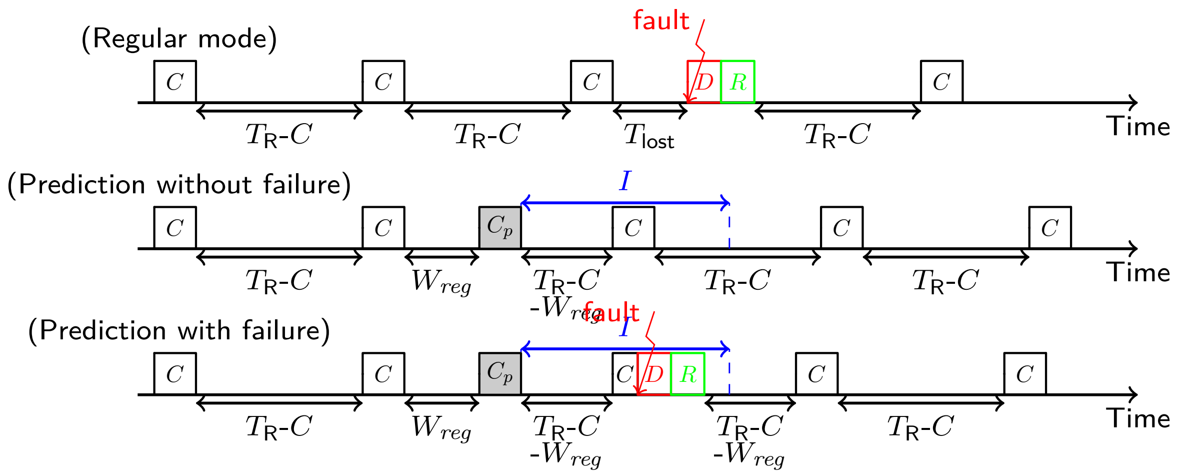


Figure 4.9: Outline of strategy INSTANT.

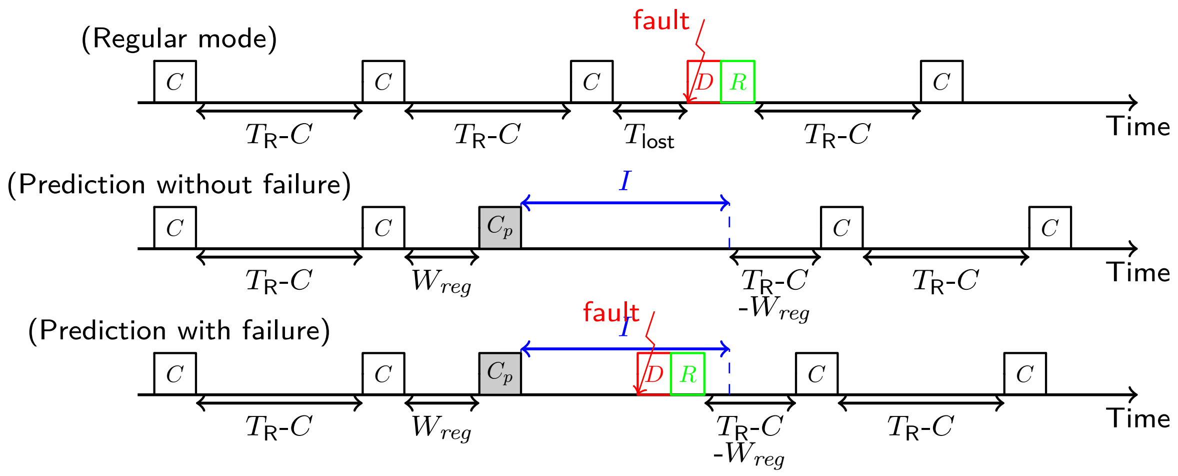


Figure 4.10: Outline of strategy NOCKPTI.

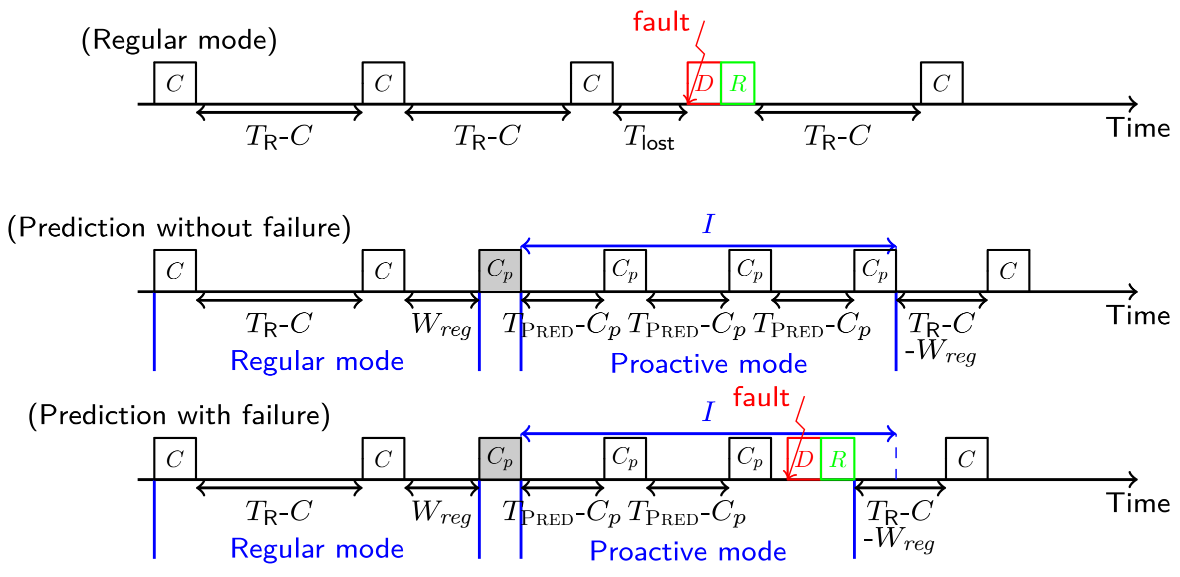


Figure 4.11: Outline of strategy WithCkptI.

$[t_0 - C_p, t_0]$. But here, we return to regular mode only at time $t_0 + I$, where we resume the work needed to complete the interrupted period of the regular mode. During the whole length of the time-window, we execute work without checkpointing, at the risk of losing work if a fault indeed strikes. But for a small value of I, it may not be worthwhile to checkpoint during the prediction window (if at all possible, since there is no choice if $I < C_p$).
(3) WithCkptI, for *With checkpoints during prediction window*– The third strategy (see Figure 4.11) is intended for a longer prediction window and assumes that $C_p \leq I$: the algorithm interrupts the current period (of scheduled length T_R), and checkpoints during the interval $[t_0 - C_p, t_0]$, but now also decides to take several checkpoints during the prediction window. The period T_{PRED} of these checkpoints in proactive mode will presumably be shorter than T_R, to take into account the higher fault probability. In the following, we analytically compute the optimal number of such periods. But we assume that there is at least one period here, hence, that we take at least one checkpoint (in the absence of faults), which implies $C_p \leq I$. We return to regular mode either right after the fault strikes within the time window $[t_0, t_0 + I]$, or at time $t_0 + I$ if no actual fault happens within this window. Then, we resume the work needed to complete the interrupted period of the regular mode. The third strategy is the most complex to describe, and the complete behavior of the corresponding scheduling algorithm is shown in Algorithm 5.

Note that, for all strategies, we insert some additional work for the particular case where there is not enough time to take a checkpoint before entering proactive mode (because a checkpoint for the regular mode is currently on-going). We account for this work as idle time in the expression of the waste, to ease the analysis. Our expression of the waste is thus an upper bound.

Strategy WithCkptI

In this subsection we evaluate the execution time under heuristic WithCkptI. To do so, we partition the whole execution into time intervals defined by the presence or absence of events.

Algorithm 5: WITHCKPTI.

```
1  if fault happens then
2      After downtime, execute recovery;
3      Enter regular mode;
4  if in proactive mode for a time greater than or equal to $I$ then
5      Switch to regular mode;
6  if Prediction made with interval $[t, t+I]$ and prediction taken into account then
7      Let $t_C$ be the date of the last checkpoint under regular mode to start no later than
       $t - C_p$;
8      if $t_C + C < t - C_p$ then (enough time for extra checkpoint)
9          Take a checkpoint starting at time $t - C_p$
10     else (no time for the extra checkpoint)
11         Work in the time interval $[t_C + C, t]$
12     $W_{reg} \leftarrow \max(0, t - C_p - (t_C + C))$ ;
13     Switch to proactive mode at time $t$;
14 while in regular mode and no predictions are made and no faults happen do
15     Work for a time $T_R$-$W_{reg}$-$C$ and then checkpoint;
16     $W_{reg} \leftarrow 0$;
17 while in proactive mode and no faults happen do
18     Work for a time $T_{\text{PRED}}$-$C_p$ and then checkpoint;
```

An interval starts and ends with either the completion of a checkpoint or of a recovery (after a failure). To ease the analysis, we make a simplifying hypothesis: we assume that *at most* one event, failure or prediction, occurs within any interval of length $T_R + I + C_p$. In particular, this implies that a prediction or an unpredicted fault always take place during the regular mode.

We list below the four types of intervals, and evaluate their respective average length, together with the average work completed during each of them (see Table 4.7 for a summary):

1. **Two consecutive regular checkpoints with no intermediate events.** The time elapsed between the completion of the two checkpoints is exactly T_R, and the work done is exactly $T_R - C$.
2. **Unpredicted fault.** Recall that, because of the simplifying hypothesis, the fault happens in regular mode. Because instants where the fault strikes and where the last checkpoint was taken are independent, on average the fault strikes at time $T_R/2$. A downtime of length D and a recovery of length R occur before the interval completes. There is no work done.
3. **False prediction.** Recall that it happens in regular mode. There are two cases:
   (a) **Taken into account.** This happens with probability q. The interval lasts $T_R + C_p + I$, since we take a proactive checkpoint and spend the time I in proactive mode. The work done is $(T_R - C) + (I - \frac{I}{T_{\text{PRED}}} C_p)$.
   (b) **Not taken into account.** This happens with probability $1 - q$. The interval lasts T_R and the work done is $T_R - C$.

   Considering both cases with their probabilities, the average time spent is equal to: $q(T_R + C_p + I) + (1-q)T_R = T_R + q(C_p + I)$. The average work done is: $q(T_R - C + I - \frac{I}{T_{\text{PRED}}} C_p) + (1-q)(T_R - Cr) = T_R - C + q(I - \frac{I}{T_{\text{PRED}}} C_p)$.
4. **True prediction.** Recall that it happens in regular mode. There are two cases:

Mode	Number of intervals	Time spent	Work done
(1)	w_1	T_{R}	$T_{\mathrm{R}} - C$
(2)	$w_2 = \frac{\textsc{Time}_{\mathrm{Final}}}{\mu_{\mathrm{NP}}}$	$T_{\mathrm{R}}/2 + D + R$	0
(3)	$w_3 = \frac{(1-p)\textsc{Time}_{\mathrm{Final}}}{\mu_{\mathrm{P}}}$	$T_{\mathrm{R}} + q(I + C_p)$	$T_{\mathrm{R}} - C + q(I - \frac{I}{T_{\textsc{Pred}}} C_p)$
(4)	$w_4 = \frac{p\textsc{Time}_{\mathrm{Final}}}{\mu_{\mathrm{P}}}$	$q(T_{\mathrm{R}} + \mathbb{E}_I^{(f)} + C_p)$ $+(1-q)T_{\mathrm{R}}/2 + D + R$	$q\left(T_{\mathrm{R}} - C + \left(\frac{\mathbb{E}_I^{(f)}}{T_{\textsc{Pred}}} - 1\right)(T_{\textsc{Pred}} - C_p)\right)$

Table 4.7: Summary of the different interval types for WithCkptI.

(a) **Taken into account.** Let $\mathbb{E}_I^{(f)}$ be the average time at which a fault occurs within the prediction window (the time at which the fault strikes is certainly correlated to the starting time of the prediction window; $\mathbb{E}_I^{(f)}$ may not be equal to $I/2$). Up to time $\mathbb{E}_I^{(f)}$, we work and checkpoint in proactive mode, with period $T_{\textsc{Pred}}$. In addition, we take a proactive checkpoint right before the start of the prediction window. Then we spend the time $\mathbb{E}_I^{(f)}$ in proactive mode, and we have a downtime and a recovery. Hence, such an interval lasts $T_{\mathrm{R}} + C_p + \mathbb{E}_I^{(f)} + D + R$ on average. The total work done during the interval is $T_{\mathrm{R}} - C + x(T_{\textsc{Pred}} - C_p)$ where x is the expectation of the number of proactive checkpoints successfully taken during the prediction window. Here, $x \approx \frac{\mathbb{E}_I^{(f)}}{T_{\textsc{Pred}}} - 1$.

(b) **Not taken into account.** On average the fault occurs at time $T_{\mathrm{R}}/2$. The time interval has duration $T_{\mathrm{R}}/2 + D + R$, and there is no work done.

Overall the time spent is $q(T_{\mathrm{R}} + C_p + \mathbb{E}_I^{(f)} + D + R) + (1-q)(T_{\mathrm{R}}/2 + D + R)$, and the work done is $q(T_{\mathrm{R}} - C + (\frac{\mathbb{E}_I^{(f)}}{T_{\textsc{Pred}}} - 1)(T_{\textsc{Pred}} - C_p)) + (1-q)0$.

We want to estimate the total execution time, $\textsc{Time}_{\mathrm{Final}}$. So far, we have evaluated the length, and the work done, for each of the interval types. We now estimate the expectation of the number of intervals of each type. Consider the intervals defined by an event whose mean time between occurrences is ν. On average, during a time T, there will be T/ν such intervals. Due to the simplifying hypothesis, intervals of different types never overlap. Table 4.7 presents the estimation of the number of intervals of each type.

To estimate the time spent within intervals of a given type, we multiply the expectation of the number of intervals of that type by the expectation of the time spent in each of them. Of course, multiplying expectations is correct only if the corresponding random variables are independent. Nevertheless, we hope that this will lead us to a good approximation of the expected execution time. We will assess the quality of the approximation through simulations in subsection 4.3.3. We have:

$$\begin{aligned}\textsc{Time}_{\mathrm{Final}} = w_1 \times T_{\mathrm{R}} + w_2\left(\frac{T_{\mathrm{R}}}{2} + D + R\right) + w_3\left(T_{\mathrm{R}} + q(I + C_p)\right)\\ + w_4\left(q(T_{\mathrm{R}} + \mathbb{E}_I^{(f)} + C_p) + (1-q)\frac{T_{\mathrm{R}}}{2} + D + R\right)\end{aligned} \quad (4.6)$$

We use the same line of reasoning to compute the overall amount of work done, that must be equal, by definition, to $\textsc{Time}_{\text{base}}$, the execution time of the application without any overhead:

$$\begin{aligned}\textsc{Time}_{\text{base}} =& w_1(T_{\text{R}} - C) + w_2 \times 0 + w_3 \left(T_{\text{R}} - C + q\left(I - \frac{I}{T_{\text{Pred}}}C_p\right)\right) \\ &+ w_4\left(q\left(T_{\text{R}} - C + \left(\frac{\mathbb{E}_I^{(f)}}{T_{\text{Pred}}} - 1\right)(T_{\text{Pred}} - C_p)\right)\right)\end{aligned} \tag{4.7}$$

This equation gives the value of w_1 as a function of the other parameters. Looking at Equations (4.6) and (4.7), and at the values of w_2, w_3, and w_4, we remark that $\textsc{Time}_{\text{Final}}$ can be rewritten as a function of q as follows: $\textsc{Time}_{\text{Final}} = \alpha\textsc{Time}_{\text{base}} + \beta\textsc{Time}_{\text{Final}} + q\gamma\textsc{Time}_{\text{Final}}$, that is $\textsc{Time}_{\text{Final}} = \frac{\alpha}{1-\beta-q\gamma}\textsc{Time}_{\text{base}}$, where neither α, nor β, nor γ depend on q. The derivative of $\textsc{Time}_{\text{Final}}$ with respect to q has constant sign. Hence, in an optimal solution, either $q = 0$ or $q = 1$. This (somewhat unexpected) conclusion is that the predictor should sometimes be always trusted, and sometimes never, but no in-between value for q will do a better job. Thus we can now focus on the two functions $\textsc{Time}_{\text{Final}}$, the one when $q = 0$ ($\textsc{Time}^{\{0\}}_{\text{Final}}$), and the one when $q = 1$ ($\textsc{Time}^{\{1\}}_{\text{Final}}$).

When $q = 0$, from Table 4.7 and Equations (4.6) and (4.7), we derive that

$$\textsc{Time}^{\{0\}}_{\text{Final}} = \frac{T_{\text{R}}}{T_{\text{R}} - C}\textsc{Time}_{\text{base}} + \frac{\textsc{Time}^{\{0\}}_{\text{Final}}}{\mu}\left(\frac{T_{\text{R}}}{2} + D + R\right), \text{ i.e., that}$$

$$\left(1 - \frac{C}{T_{\text{R}}}\right)\left(1 - \frac{T_{\text{R}}/2 + D + R}{\mu}\right)\textsc{Time}^{\{0\}}_{\text{Final}} = \textsc{Time}_{\text{base}} \tag{4.8}$$

This is exactly the equation from Section 4.2 in the case of exact-date predictions that are never taken into account (a good sanity check!). When $q = 1$, we have:

$$\begin{aligned}\textsc{Time}^{\{1\}}_{\text{Final}} =& \textsc{Time}_{\text{base}}\frac{T_{\text{R}}}{T_{\text{R}} - C} \\ &- \frac{\textsc{Time}^{\{1\}}_{\text{Final}}}{\mu_{\text{P}}}\frac{T_{\text{R}}}{T_{\text{R}}-C}\left((T_{\text{R}}-C)+(1-p)\left(I-\frac{I}{T_{\text{Pred}}}C_p\right)+p\left(\frac{\mathbb{E}_I^{(f)}}{T_{\text{Pred}}}-1\right)(T_{\text{Pred}}-C_p)\right) \\ &+ \frac{\textsc{Time}^{\{1\}}_{\text{Final}}}{\mu_{\text{NP}}}\left(\frac{T_{\text{R}}}{2} + D + R\right) + \frac{(1-p)\textsc{Time}^{\{1\}}_{\text{Final}}}{\mu_{\text{P}}}(T_{\text{R}} + I + C_p) \\ &+ \frac{p\textsc{Time}^{\{1\}}_{\text{Final}}}{\mu_{\text{P}}}\left(T_{\text{R}} + C_p + \mathbb{E}_I^{(f)} + D + R\right)\end{aligned}$$

After a little rewriting we obtain:

$$\begin{aligned}\frac{\textsc{Time}_{\text{base}}}{\textsc{Time}^{\{1\}}_{\text{Final}}} =& \frac{r}{p\mu}\left(1 - \frac{C_p}{T_{\text{Pred}}}\right)\left((1-p)I + p\left(\mathbb{E}_I^{(f)} - T_{\text{Pred}}\right)\right) \\ &+\left(1 - \frac{C}{T_{\text{R}}}\right)\left(1 - \frac{1}{p\mu}\left(p(D+R) + rC_p + (1-r)p\frac{T_{\text{R}}}{2} + r\left((1-p)I+p\mathbb{E}_I^{(f)}\right)\right)\right)\end{aligned}$$

Finally, the waste is equal by definition to $\frac{\textsc{Time}_{\text{Final}}-\textsc{Time}_{\text{base}}}{\textsc{Time}_{\text{Final}}}$. Therefore, we have:

$$\begin{aligned}\textsc{Waste} =& 1 - \frac{r}{p\mu}\left(1 - \frac{C_p}{T_{\text{Pred}}}\right)\left((1-p)I + p\left(\mathbb{E}_I^{(f)} - T_{\text{Pred}}\right)\right) \\ &-\left(1-\frac{C}{T_{\text{R}}}\right)\left(1-\frac{1}{p\mu}\left(p(D+R)+rC_p+(1-r)p\frac{T_{\text{R}}}{2}+r\left((1-p)I+p\mathbb{E}_I^{(f)}\right)\right)\right)\end{aligned} \tag{4.9}$$

Mode	Number of intervals	Time spent	Work done
(1)	w_1	T_R	$T_R - C$
(2)	$w_2 = \frac{\text{TIME}_{\text{Final}}}{\mu_{NP}}$	$T_R/2 + D + R$	0
(3)	$w_3 = \frac{(1-p)\text{TIME}_{\text{Final}}}{\mu_P}$	$T_R + q(I + C_p)$	$T_R - C + qI$
(4)	$w_4 = \frac{p\text{TIME}_{\text{Final}}}{\mu_P}$	$q(T_R + \mathbb{E}_I^{(f)} + C_p)$ $+(1-q)T_R/2 + D + R$	$q\left(T_R - C\right)$

Table 4.8: Summary of the different interval types for NoCkptI.

Waste minimization When $q = 0$, the optimal period can readily be computed from Equation (4.8) and we derive that the optimal period is $\sqrt{2(\mu - (D+R))C}$. This defines a periodic policy we call RFO, for Refined First-Order approximation. We now minimize the waste of the strategy where $q = 1$. In order to compute the optimal value for T_{PRED}, we identify the fraction of the waste in Equation (4.9) that depends on T_{PRED}. We can rewrite Equation (4.9) as:

$$\text{WASTE}^{\{1\}} = \alpha + \frac{r}{p\mu}\left(\left((1-p)\,I + p\mathbb{E}_I^{(f)}\right)\frac{C_p}{T_{\text{PRED}}} + pT_{\text{PRED}}\right)$$

where α does not depend on T_{PRED}. The waste is thus minimized when T_{PRED} is equal to $T_{\text{PRED}}^{\text{extr}} = \sqrt{\frac{\left((1-p)I+p\mathbb{E}_I^{(f)}\right)C_p}{p}}$. Note that we always have to enforce that T_{PRED} is larger than C_p and does not exceed I. Therefore, the optimal period $T_{\text{PRED}}^{\text{opt}}$ is defined as follows: $T_{\text{PRED}}^{\text{opt}} = \min\{I,\ \max\{C_p,\ T_{\text{PRED}}^{\text{extr}}\}\}$. The rounding only occurs for extreme cases.

In order to compute the optimal value for T_R, we identify the fraction of the waste in Equation (4.9) that depends on T_R. We can rewrite Equation (4.9) as:

$$\text{WASTE}^{\{1\}} \quad = \quad \beta \ + \ \frac{C}{T_R}\left(1-\frac{1}{p\mu}\left(p(D+R)+r\left(C_p+(1-p)\,I+p\mathbb{E}_I^{(f)}\right)\right)\right) \ + \ \frac{1-r}{\mu}\frac{T_R}{2} \tag{4.10}$$

where β does not depend on T_R because $T_{\text{PRED}}^{\text{opt}}$ does not depend on T_R. Therefore, $\text{WASTE}^{\{1\}}$ is minimized when T_R is equal to

$$T_R^{\text{extr}} = \sqrt{\frac{2C\left(p\mu - \left(p(D+R) + r\left(C_p + \left((1-p)\,I + p\mathbb{E}_I^{(f)}\right)\right)\right)\right)}{p(1-r)}} \tag{4.11}$$

Recall that we must always enforce that T_R^{opt} is always greater than C. Also, note that when $r = 0$, we do obtain the same period as without a predictor. Finally, if we assume that, on average, fault strikes at the middle of the prediction window, i.e., $\mathbb{E}_I^{(f)} = \frac{I}{2}$, we obtain simplified values:

$$T_{\text{PRED}}^{\text{extr}} = \sqrt{\frac{(2-p)IC_p}{p}} \text{ and } T_R^{\text{extr}} = \sqrt{\frac{2C\left(p\mu - \left(p(D+R) + r\left(C_p + \left(1 - \frac{p}{2}\right)I\ \right)\right)\right)}{p(1-r)}}$$

Strategy NoCkptI

In this subsection we evaluate the execution time under heuristic NoCkptI. For clarity, we only summarize results and refer to [RR3] for details. The analysis is similar to that for

Mode	Number of intervals	Time spent	Work done
(1)	w_1	T_{R}	$T_{\mathrm{R}} - C$
(2)	$w_2 = \frac{\mathrm{TIME}_{\mathrm{Final}}}{\mu_{\mathrm{NP}}}$	$T_{\mathrm{R}}/2 + D + R$	0
(3)	$w_3 = \frac{(1-p)\mathrm{TIME}_{\mathrm{Final}}}{\mu_{\mathrm{P}}}$	$T_{\mathrm{R}} + qC_p$	$T_{\mathrm{R}} - C$
(4)	$w_4 = \frac{p\mathrm{TIME}_{\mathrm{Final}}}{\mu_{\mathrm{P}}}$	$q(T_{\mathrm{R}} + \mathbb{E}_I^{(f)} + C_p)$ $+(1-q)T_{\mathrm{R}}/2 + D + R$	$q\,(T_{\mathrm{R}} - C)$

Table 4.9: Summary of the different interval types for INSTANT.

WITHCKPTI. Table 4.8 provides the estimation of the number of intervals of each type. As for WITHCKPTI, one shows that in an optimal solution, either $q = 0$ or $q = 1$. When $q = 0$, we derive that

$$\mathrm{TIME}_{\mathrm{Final}}^{\{0\}} = \frac{T_{\mathrm{R}}}{T_{\mathrm{R}} - C}\mathrm{TIME}_{\mathrm{base}} + \frac{\mathrm{TIME}_{\mathrm{Final}}^{\{0\}}}{\mu}\left(\frac{T_{\mathrm{R}}}{2} + D + R\right), \text{ i.e., that}$$

$$\left(1-\frac{C}{T_{\mathrm{R}}}\right)\left(1 - \frac{T_{\mathrm{R}}/2 + D + R}{\mu}\right)\mathrm{TIME}_{\mathrm{Final}}^{\{0\}} = \mathrm{TIME}_{\mathrm{base}} \tag{4.12}$$

This is exactly the equation from Section 4.2 in the case of exact-date predictions that are never taken into account, what we had already retrieved with WITHCKPTI (same sanity check!). When $q = 1$, we derive that:

$$\mathrm{WASTE} = 1 - \frac{r}{p\mu}(1-p)I - \left(1-\frac{C}{T_{\mathrm{R}}}\right)\times$$
$$\left(1-\frac{1}{p\mu}\left(p(D+R)+rC_p+(1-r)p\frac{T_{\mathrm{R}}}{2}+r\left((1-p)I+p\mathbb{E}_I^{(f)}\right)\right)\right) \tag{4.13}$$

Waste minimization The waste is minimized as follows:
— When $q = 0$, the optimal value for T_{R} is the same as the one we computed for WITHCKPTI in the case $q = 0$.
— When $q = 1$, the value of T_{R} that minimizes the waste is $T_{\mathrm{R}}^{\mathrm{extr}}$, the value given by Equation (4.11).

Strategy Instant

In this subsection we evaluate the execution time under heuristic NOCKPTI. For clarity, we only summarize results and refer to [RR3] for details. The analysis is similar to the previous ones. Table 4.9 provides the estimation of the number of intervals of each type. As before, one shows that in an optimal solution, either $q = 0$ or $q = 1$. When $q = 0$, we derive, once again, that

$$\mathrm{TIME}_{\mathrm{Final}}^{\{0\}} = \frac{T_{\mathrm{R}}}{T_{\mathrm{R}} - C}\mathrm{TIME}_{\mathrm{base}} + \frac{\mathrm{TIME}_{\mathrm{Final}}^{\{0\}}}{\mu}\left(\frac{T_{\mathrm{R}}}{2} + D + R\right), \text{ i.e., that}$$

$$\left(1-\frac{C}{T_{\mathrm{R}}}\right)\left(1 - \frac{T_{\mathrm{R}}/2 + D + R}{\mu}\right)\mathrm{TIME}_{\mathrm{Final}}^{\{0\}} = \mathrm{TIME}_{\mathrm{base}} \tag{4.14}$$

This is exactly the equation from Section 4.2 in the case of exact-date predictions that are never taken into account, what we had already remarked with WITHCKPTI and NOCKPTI (yet another good sanity check!). When $q = 1$, we obtain

$$\text{WASTE} = 1 - \left(1 - \frac{C}{T_\text{R}}\right) \times \left(1 - \frac{1}{p\mu}\left(p(D+R) + rC_p + (1-r)p\frac{T_\text{R}}{2} + pr\mathbb{E}_I^{(f)}\right)\right) \quad (4.15)$$

Waste minimization The waste is minimized as follows:

— When $q = 0$, the optimal value for T_R is the same as for WITHCKPTI and for NOCKPTI in the case $q = 0$.

— When $q = 1$, the optimal value for T_R is

$$T_\text{R}^\text{extr} = \sqrt{\frac{2C\left(p\mu - \left(p(D+R) + rC_p + pr\mathbb{E}_I^{(f)}\right)\right)}{p(1-r)}}$$

Again, recall that we must always enforce that T_R^opt is always greater than C. Finally, if we assume that, on average, fault strikes at the middle of the prediction window, i.e., $\mathbb{E}_I^{(f)} = \frac{I}{2}$, we have:

$$T_\text{R}^\text{extr} = \sqrt{\frac{2C\left(p\mu - \left(p(D+R) + rC_p + pr\frac{I}{2}\right)\right)}{p(1-r)}}.$$

4.3.3 Simulation results

An experimental validation of the models at targeted scale would require running a large application several times, for each checkpointing strategy, for each fault predictor, and for each platform size. This would require a prohibitive amount of computational hours. Furthermore, some of the targeted platform sizes currently exist only as reasonable projections. Therefore, we resort to simulations. We present the simulation framework in subsection 4.3.3. Then we report results using the characteristics of two fault predictors (subsection 4.3.3). Additional figures and data results are available in [RR3].

Simulation framework

In order to validate the model, we have instantiated it with several scenarios. The simulations use parameters that are representative of current and forthcoming large-scale platforms [68, 69]. We take $C = R = 600$ seconds, and $D = 60$ seconds. We consider three scenarios where proactive checkpoints are (i) exactly as expensive as periodic checkpoints ($C_p = C$); (ii) ten times cheaper ($C_p = 0.1C$); and (iii) two times more expensive ($C_p = 2C$). The individual (processor) MTBF is $\mu_\text{ind} = 125$ years, and the total number of processors N varies from $N = 2^{16} = 16,384$ to $N = 2^{19} = 524,288$, so that the platform MTBF μ varies from $\mu = 4,010$ min (about 2.8 days) down to $\mu = 125$ min (about 2 hours). For instance the Jaguar platform, with $N = 45,208$ processors, is reported to have experienced about one fault per day [46], which leads to $\mu_\text{ind} = \frac{45,208}{365} \approx 125$ years. The application size is set to $\text{TIME}_\text{base} = 10,000$ years/N.

We use Maple to analytically compute and plot the optimal value of the waste for the three prediction-aware policies, INSTANT, NOCKPTI, and WITHCKPTI, for the prediction-ignoring policy RFO (corresponding to the case $q = 0$), and for the reference heuristic DALY (Daly's [54] periodic policy). In order to check the accuracy of our model, we have compared the analytical results with results obtained with a discrete-event simulator. The simulation engine generates a random trace of faults, parameterized either by an Exponential fault distribution or by Weibull distribution laws with shape parameter 0.5 or 0.7. Note that Exponential faults are widely used for theoretical studies, while Weibull faults are representative of the behavior of real-world platforms [64, 63, 66]. In both cases, the distribution is scaled so that its expectation corresponds to the platform MTBF μ. With probability r, we decide if a fault is predicted or not. The simulation engine also generates a random trace of false predictions, whose distribution is identical to that of the first trace (results are similar when false predictions follow a uniform distribution [RR3]). This second distribution is scaled so that its expectation is equal to $\frac{\mu_P}{1-p} = \frac{p\mu}{r(1-p)}$, the inter-arrival time of false predictions. Finally, both traces are merged to produce the final trace including all events (true predictions, false predictions, and non predicted faults). The source code of the fault-simulator and the raw simulation results are freely available [72]. Each reported value is the average over 100 randomly generated instances.

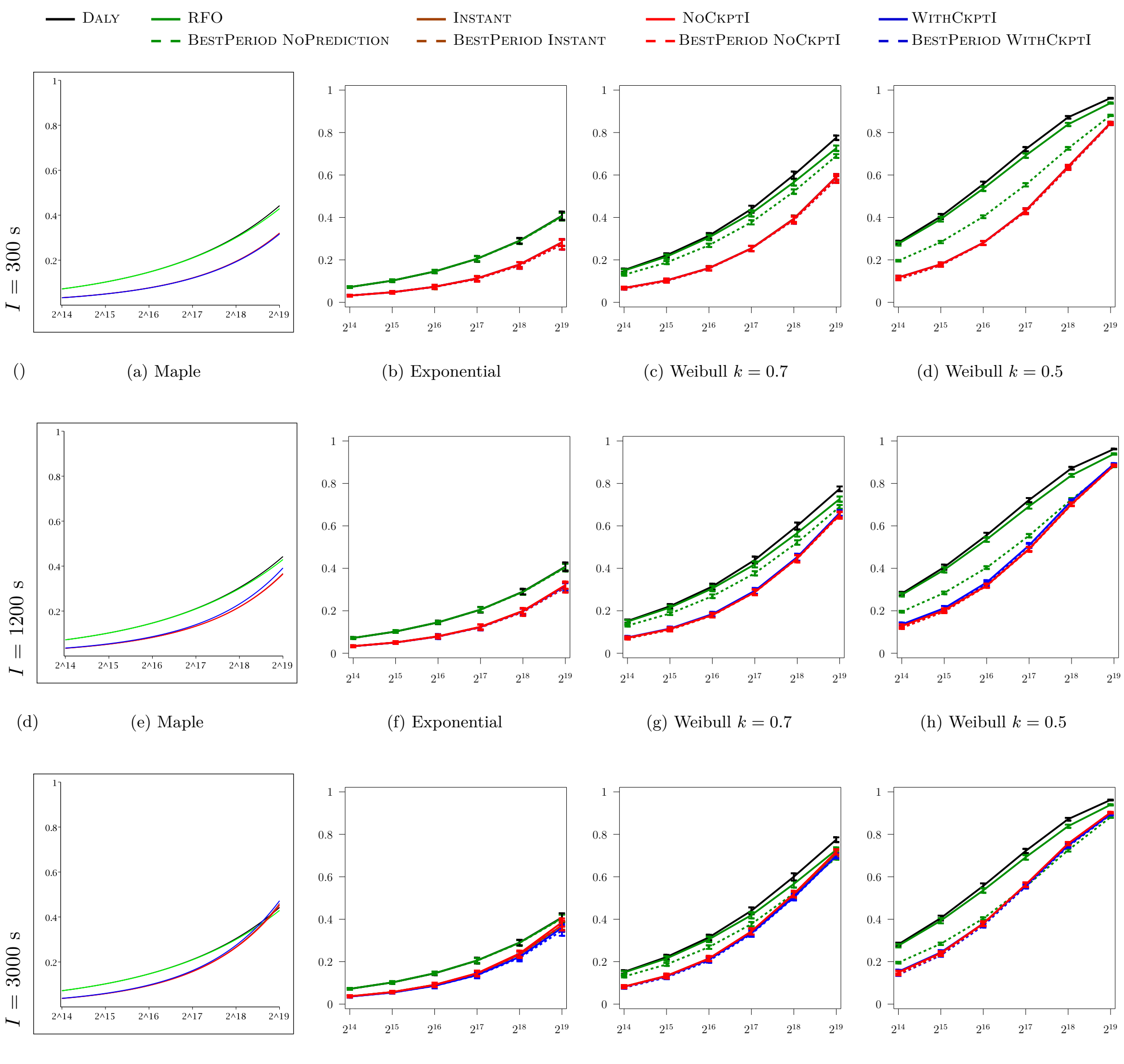
In the simulations, we compare the five checkpointing strategies listed above. To assess the quality of each strategy, we compare it with its BESTPERIOD counterpart, defined as the same strategy but using the best possible period T_R. This latter period is computed via a brute-force numerical search for the optimal period. Altogether, there are four BESTPERIOD heuristics, one for each of the three variants with prediction, and one for the case where we ignore predictions, which corresponds to both DALY and RFO. Altogether we have a rich set of nine heuristics, which enables us to comprehensively assess the actual quality of the proposed strategies. Note that for computer algebra plots, obviously we do not need BESTPERIOD heuristics, since each period is already chosen optimally from the equations.

We experiment with two predictors from the literature: one accurate predictor with high recall and precision [51], namely with $p = 0.82$ and $r = 0.85$, and another predictor with more limited recall and precision [52], namely with $p = 0.4$ and $r = 0.7$. In both cases, we use five different prediction windows, of size $I = 300$, 600, 900, 1200, and 3000 seconds. Figure 4.12 shows the average waste degradation of the nine heuristics for both predictors, as a function of the number of processors N. We draw the plots as a function of the number of processors N rather than of the platform MTBF $\mu = \mu_{ind}/N$, because it is more natural to see the waste increase with larger platforms; however, this work is agnostic of the granularity of the processors and intrinsically focuses on the impact of the MTBF on the waste.

Analysis of the results

We start with a preliminary remark: when the graphs for INSTANT and WITHCKPTI cannot be seen in the figures, this is because their performance is identical to that of NOCKPTI, and their respective graphs are superposed.

We first compare the analytical results, plotted by the Maple curves, to the simulations results. As shown in Figure 4.12, there is a good correspondence between the analytical curves and the simulations, especially those using an Exponential distribution of failures. However, the larger the platform (or the smaller the MTBF), the less realistic our assumption that no two events happen during an interval of length $T_R + I + C_p$, and the analytical models become less accurate for prediction-aware heuristics. Therefore, the analytical results are overly pessimistic in the most failure-prone platforms. Also, recall that an exponential law is a Weibull law of shape parameter 1. Therefore, the further the distribution of failures is from an exponential

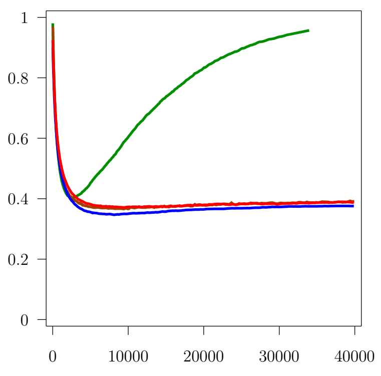


() (a) Maple (b) Exponential (c) Weibull $k = 0.7$ (d) Weibull $k = 0.5$

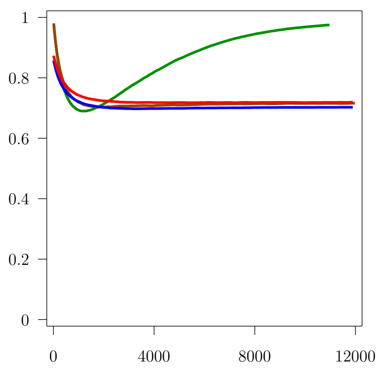
(d) (e) Maple (f) Exponential (g) Weibull $k = 0.7$ (h) Weibull $k = 0.5$

(h) (i) Maple (j) Exponential (k) Weibull $k = 0.7$ (l) Weibull $k = 0.5$

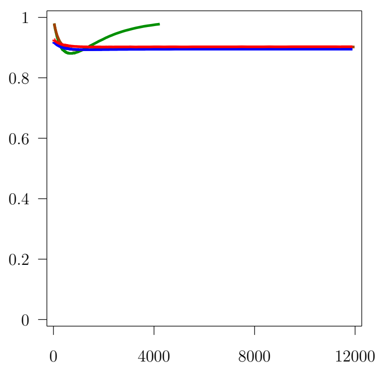
Figure 4.12: Waste as a function of number N of processors, when $p = 0.82$, $r = 0.85$, $C_p = C$.



(a) Exponential

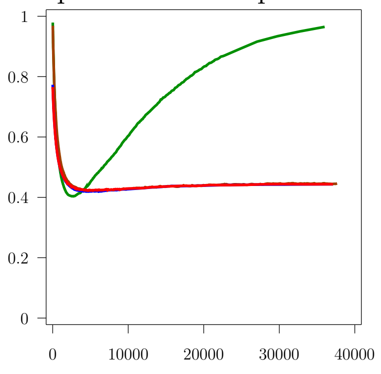


(b) Weibull $k = 0.7$

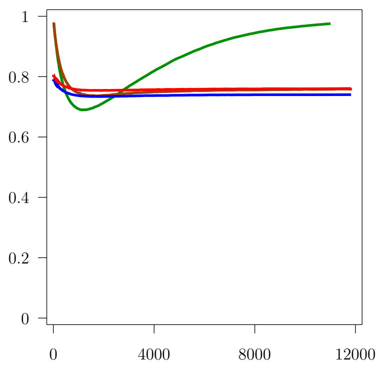


(c) Weibull $k = 0.5$

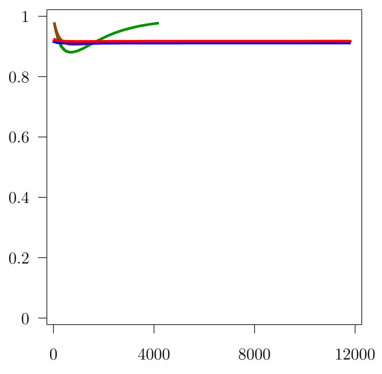
Figure 4.13: Waste as function of the period T_{R}, with $p = 0.82$, $r = 0.85$, $C_p = C$, $I = 3000s$, and a platform of 2^{19} processors.



(a) Exponential



(b) Weibull $k = 0.7$



(c) Weibull $k = 0.5$

Figure 4.14: Waste as function of the period T_{R}, with $p = 0.4$, $r = 0.7$, $C_p = C$, $I = 3000s$, and a platform of 2^{19} processors.

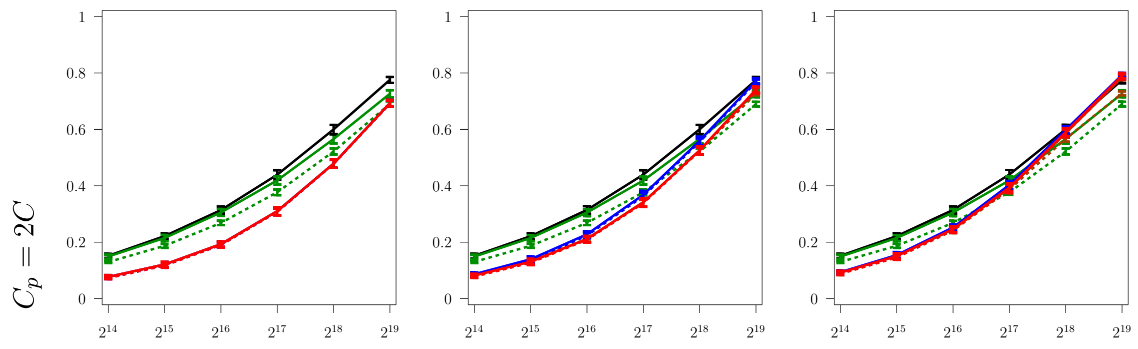
law, the larger the difference between analytical results and simulated ones. However, in all cases, the analytical results are able to predict the general *trends*.

A second assessment of the quality of our analysis comes from the BESTPERIOD variants of our heuristics. When predictions are not taken into account, DALY, and to a lesser extent RFO, are not close to the optimal period given by BESTPERIOD (a similar observation was made in [61]). This gap increases when the distribution is further apart from an Exponential distribution. However, prediction-aware heuristics are very close to BESTPERIOD in almost all configurations. The only exception is with heuristics INSTANT when $C_p = 2C$, the total number of processors N is equal to either 2^{18} and 2^{19}, and I is large. However, when I = 3000s and $N = 2^{19}$, the platform MTBF is approximately equal to $6C_p$ which renders our hypothesis and analysis invalid. The difference in this case between INSTANT and its BESTPERIOD should therefore not come as a surprise.

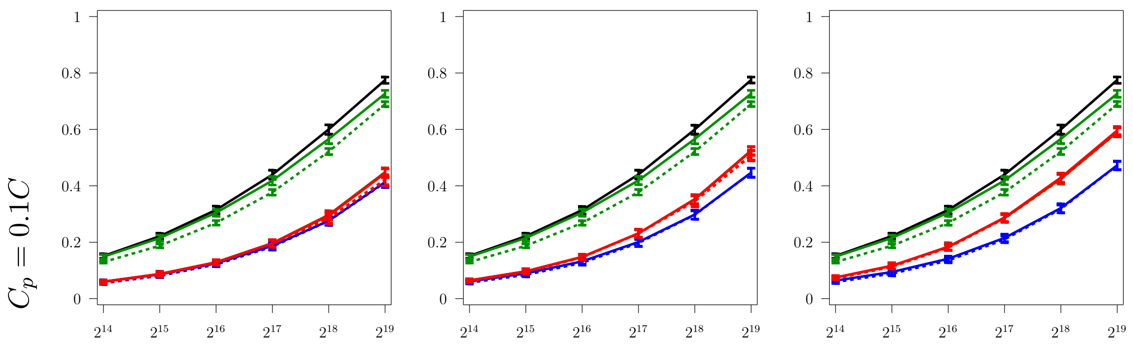
To better understand why close-to-optimal periods are obtained by prediction-aware heuristics (while this is not the case without predictions), we plot the waste as a function of the period T_{R} for RFO and the prediction-aware heuristics (Figure 4.14). On these figures one can see that, whatever the configuration, periodic checkpointing policies (ignoring predictions) have well-defined global optimum. (One should nevertheless remark that the performance is almost constant in the neighborhood of the optimal period, which explains why policies using different periods can obtain in practice similar performance, as in [77].) For prediction-aware heuristics, however, the behavior is quite different and two scenarios are possible. In the first one, once the optimum is reached, the waste very slowly increases to reach an asymptotic value which is close to the optimum waste (e.g., when the platform MTBF is large and failures follow an exponential distribution). Therefore, any period chosen close to the optimal one, or greater than it, will deliver good quality performance. In the second scenario, the waste decreases until the period becomes larger than the application size, and the waste stays constant. In other words, in these configurations, periodic checkpointing is unnecessary, only proactive actions matter! This striking result can be explained as follows: a significant fraction of the failures are predicted, and thus taken care of, by proactive checkpoints. The impact of unpredicted failures is mitigated by the proactive measures taken for false predictions. To further mitigate the impact of unpredicted faults, the period T_{R} should be significantly shorter than the mean-time between proactive checkpoints, which would induce a lot of waste due to unnecessary checkpoints if the mean-time between unpredicted faults is large with respect to the mean-time between predictions. This greatly restrict the scenarios for which the periodic checkpointing can lead to a significant decrease of the waste.

Figure 4.15 and 4.16 presents a comparison of the checkpointing strategies for different values of C_p and I. When the prediction window I is shorter than the duration C_p of a proactive checkpoint (i.e., when $I = 300$ s and $C_p \geq C = 600$ s), there is no difference between NoCkptI and WithCkptI. When I is small but greater than C_p (say, when I is around $2C_p$), WithCkptI spends most of the prediction window taking a proactive checkpoint and NoCkptI is more efficient. When I becomes "large" with respect to C_p, WithCkptI can become more efficient than NoCkptI, but becomes significantly more efficient only if the proactive checkpoints are significantly shorter than regular (see also Table 4.10). Instant can hardly be seen in the graphs as its performance is most of the time equivalent to that of NoCkptI.

As expected, the smaller the prediction window, the more efficient the prediction-aware heuristics. Also, the smaller the number of processors (or the larger the platform MTBF), the larger the impact of the size of the prediction window. A surprising result is that taking prediction into account is not always beneficial! The analytical results predict that prediction-aware heuristics would achieve worse performance than periodic policies in our settings, as soon as the platform includes 2^{18} processors. In simulations, results are not so extreme. For the largest platforms considered, using predictions has almost no impact on performance. But when the prediction window is very large, taking predictions into account can indeed be detrimental. These observations can be explained as follows. When the platform includes 2^{19} processors, the platform MTBF is equal to 7500s. Therefore, any interval of duration 3000s has a 40% chance to include a failure: a prediction window of 3000s is not very informative, unless the precision and recall of the predictor are almost equal to 1 (which is never the case in practice). Because the predictor brings almost no knowledge, trusting it may be detrimental. When comparing the performance of, say, NoCkptI for the two predictors, one can see that when failures follow a Weibull distribution with shape parameter $k = 0.7$, $I = 600$s, and $N = 2^{18}$, NoCkptI achieves better performance than RFO when $r = 0.85$ and $p = 0.82$, but worse when $p = 0.4$ and $r = 0.7$. The latter predictor generates more false predictions —each one inducing an unnecessary proactive checkpoint— and misses more actual failures —each one destroying some work. The drawbacks of trusting the predictor outweigh the advantages. If failures are few and

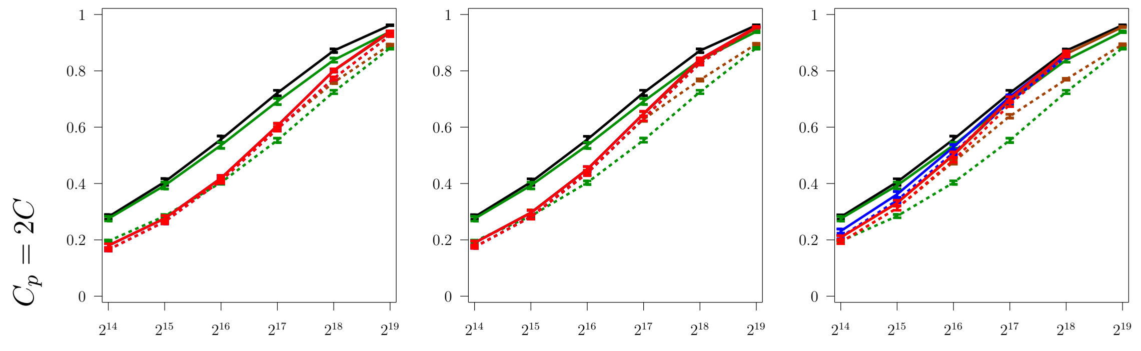


() (a) $I = 300$ s (b) $I = 1200$ s (c) $I = 3000$ s

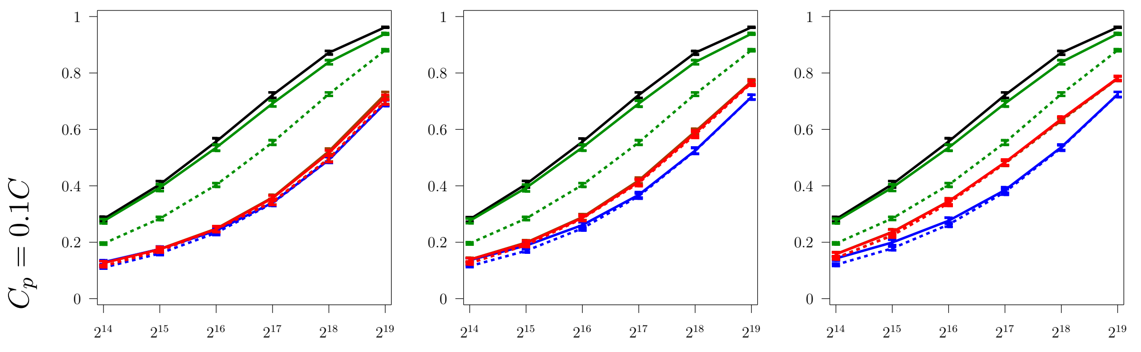


(c) (d) $I = 300$ s (e) $I = 1200$ s (f) $I = 3000$ s

Figure 4.15: Waste with $p = 0.82$, $r = 0.85$, and Weibull law of parameter 0.7.



() (a) $I = 300$ s (b) $I = 1200$ s (c) $I = 3000$ s



(c) (d) $I = 300$ s (e) $I = 1200$ s (f) $I = 3000$ s

Figure 4.16: Waste with $p = 0.4$, $r = 0.7$, and Weibull law of parameter 0.5.

apart, almost any predictor will be beneficial. When the platform MTBF is small with respect to the cost of proactive checkpoints, only almost perfect predictors will be worth using. For each set of predictor characteristics, there is a threshold for the platform MTBF under which predictions will be useless or detrimental, but above which predictions will be beneficial.

In order to compare the impact of the heuristics ignoring predictions to those using them, we report job execution times in Table 4.10 when failures follow a Weibull law of parameter 0.7. For each setting, the best performance is presented in bold if it is achieved by a prediction-aware heuristics. For the strategies with prediction, we compute the gain (expressed in percentage) over DALY, the reference strategy without prediction. We first remark that RFO achieves lower makespans than DALY with gains ranging from 1% with 2^{16} processors to 18% with 2^{19} processors. Overall, the gain due to the predictions decreases when the size of the prediction window increases, and increases with the platform size. This gain is obviously closely related to the characteristics of the predictor. When $I = 300$s, the three prediction-aware strategies are identical. When I increases, NOCKPTI achieves slightly better results than INSTANT. For low values of I, WITHCKPTI is the worst prediction-aware heuristics. But when I becomes large and if the predictor is efficient, then WITHCKPTI becomes the heuristics of choice ($I = 3000$s, $p = 0.82$, and $r = 0.85$). The reductions in the application executions times due to the predictor can be very significant. With $p = 0.85$ and $r = 0.82$ and $I = 3000$s, we save 25% of the total time with $N = 2^{19}$, and 13% with $N = 2^{16}$ using strategy WITHCKPTI. With $I = 300$s, we save up to 45% with $N = 2^{19}$, and 18% with $N = 2^{16}$ using any strategy (though NOCKPTI is slightly better than INSTANT). Then, with $p = 0.4$ and $r = 0.7$, we still save 33% of the execution time when $I = 300$s and $N = 2^{19}$, and 14% with $N = 2^{16}$. The gain gets smaller with $I = 3000$s and $N = 2^{16}$ but remains non negligible since we can save 8%. When $I = 3000$s and $N = 2^{19}$, however, the best solution is to ignore predictions and simply use RFO (we fall-back to the case $q = 0$). If we now consider a Weibull law with shape parameter 0.5 instead of 0.7 (see Table 4.11), keeping all other parameters identical ($I = 3000$s, $N = 2^{19}$, $p = 0.4$ and $r = 0.7$), then the heuristics of choice is WITHCKPTI and the gain with respect to DALY is 57.9%.

	$I = 300$ s		$I = 1200$ s		$I = 3000$ s	
	2^{16} procs	2^{19} procs	2^{16} procs	2^{19} procs	2^{16} procs	2^{19} procs
DALY	81.3	31.0	81.3	31.0	81.3	31.0
RFO	80.2 (1%)	25.5 (18%)	80.2 (1%)	25.5 (18%)	80.2 (1%)	25.5 (18%)
			$p = 0.82$, $r = 0.85$			
INSTANT	66.5 (18%)	**17.0** (45%)	68.0 (16%)	20.3 (34%)	70.9 (13%)	24.1 (22%)
NOCKPTI	**66.4** (18%)	**17.0** (45%)	**67.9** (16%)	**20.2** (35%)	71.0 (13%)	24.7 (20%)
WITHCKPTI	**66.4** (18%)	**17.0** (45%)	68.3 (16%)	20.6 (33%)	**70.6** (13%)	**23.1** (25%)
			$p = 0.4$, $r = 0.7$			
INSTANT	70.3 (13%)	20.9 (33%)	72.0 (11%)	24.6 (21%)	**75.0** (8%)	27.7 (11%)
NOCKPTI	**70.2** (14%)	**20.6** (33%)	**71.8** (12%)	**24.2** (22%)	**75.0** (8%)	28.7 (7%)
WITHCKPTI	**70.2** (14%)	**20.6** (33%)	73.6 (9%)	25.5 (18%)	75.1 (8%)	26.6 (14%)

Table 4.10: Job execution times (in days) under the different checkpointing policies, when failures follow a Weibull distribution of shape parameter 0.7. Gains are reported with respect to DALY.

	$I = 300$ s		$I = 1200$ s		$I = 3000$ s	
	2^{16} procs	2^{19} procs	2^{16} procs	2^{19} procs	2^{16} procs	2^{19} procs
DALY	125.7	185.0	125.7	185.0	125.7	185.0
RFO	120.1 (4%)	114.8 (38%)	120.1 (4%)	114.8 (38%)	120.1 (4%)	114.8 (38%)
$p = 0.82$, $r = 0.85$						
INSTANT	**77.4** (38%)	45.2 (76%)	82.0 (35%)	60.8 (67%)	**89.7** (29%)	70.6 (62%)
NOCKPTI	**77.4** (38%)	**44.9** (76%)	**81.8** (35%)	**60.7** (67%)	90.0 (28%)	71.5 (61%)
WITHCKPTI	**77.4** (38%)	**44.9** (76%)	83.6 (33%)	64.4 (65%)	89.8 (29%)	**66.2** (64%)
$p = 0.4$, $r = 0.7$						
INSTANT	84.5 (33%)	59.6 (68%)	89.4 (29%)	76.6 (58%)	**97.7** (22%)	81.9 (56%)
NOCKPTI	**84.4** (33%)	**58.3** (68%)	**89.1** (29%)	76.8 (58%)	97.9 (22%)	83.7 (55%)
WITHCKPTI	**84.4** (33%)	**58.3** (68%)	93.8 (25%)	**75.4** (59%)	97.8 (22%)	**77.7** (58%)

Table 4.11: Job execution times (in days) under the different checkpointing policies, when failures follow a Weibull distribution of shape parameter 0.5. Gains are reported with respect to DALY.

4.4 Conclusion

In this work, we have studied the impact of fault prediction on checkpointing strategies. The importance of good prediction techniques is increasing with the advent of exascale platforms, for which current checkpointing techniques will not provide efficient solutions any longer [91, 92].

We have studied the impact of a predictor with exact prediction dates on periodic checkpointing strategies. We have proven that the optimal approach is to never trust the predictor in the beginning of a regular period, and to always trust it in the end of the period; the cross-over point $\frac{C_p}{p}$ depends on the time to take a proactive checkpoint and on the precision of the predictor. This striking result is somewhat unexpected, as one might have envisioned more trust regimes, with several intermediate trust levels smoothly evolving from a "never trust" policy to an "always trust" one. We have conducted simulations involving synthetic failure traces following either an Exponential distribution law or a Weibull one. We have also used log-based failure traces. In addition, we have used exact prediction dates. Through this extensive experiment setting, we have established the accuracy of the model, of its analysis, and of the predicted period (in the presence of an exact fault predictor). These simulations also show that even a not-so-good exact fault predictor can lead to quite a significant decrease in the application execution time. We have also shown that the most important characteristic of an exact fault predictor is its recall (the percentage of actually predicted faults) rather than its precision (the percentage of predictions that actually correspond to faults): *better safe than sorry*, or better prepare for a false event than miss an actual failure!

We have also studied the impact of a predictor with prediction windows on periodic checkpointing strategies. We have designed several heuristics that decide whether to trust predictions or not, when it is worth taking preventive checkpoints, and at which rate. We have introduced an analytical model to capture the waste incurred by each strategy, and provided a closed-form formula for each optimization problem, giving the optimal solution. Contrarily to the cases without prediction, or with exact date predictions, the computation of the waste requires a sophisticated analysis of the various events, including the time spent in the regular and proactive modes. We have proposed a model that is quite accurate and its validity goes beyond the conservative assumption that requires a single event per time interval; even more surprising, the accuracy of the model for prediction-aware strategies is much better than for the case without predictions, where Daly can be far from the optimal period in the case of Weibull failure distributions [61]. We have also conducted simulations that fully validate the model, and the brute-force computation of the optimal period guarantees that our prediction-aware strategies are always very close to the optimal. This holds true both for Exponential and Weibull failure distributions. With the analytical computations and the simulations, we are able to characterize when prediction is useful, and which strategy performs better, given the key parameters of the system: recall r, precision p, size of the prediction window I, size of proactive checkpoints C_p versus regular checkpoints C, and platform MTBF μ.

Altogether, the analytical model and the comprehensive results provided in this work enable to fully assess the impact of fault prediction on (optimal) checkpointing strategies.

Future work will be devoted to refine the assessment of the usefulness of prediction with trace-based failures and prediction logs from current large-scale supercomputers. Another direction for future work would be to study the impact of fault prediction on uncoordinated or hierarchical checkpointing protocols.

Chapter 5

Combining Silent Error Detection and Coordinated Checkpointing

5.1 Introduction

This work is motivated by a recent paper by Lu, Zheng and Chien [74], who introduce a *multiple checkpointing model* to compute the optimal checkpointing period with error detection latency. More precisely, Lu, Zheng and Chien [74] deal with the following problem: given errors whose inter arrival times X_e follow an Exponential probability distribution of parameter λ_e, and given error detection times X_d that follow an Exponential probability distribution of parameter λ_d, what is the optimal checkpointing period T_{opt} in order to minimize the total execution time? The problem is illustrated on Figure 5.1: the error is detected after a (random) time X_d, and one has to rollback up to the last checkpoint that precedes the occurrence of the error. Let k be the number of checkpoints that can be simultaneously kept in memory. Lu, Zheng and Chien [74] derive a formula for the optimal checkpointing period T_{opt} in the (simplified) case where k is unbounded ($k = \infty$), and they propose some numerical simulations to explore the case where k is a fixed constant.

The first major contribution of this Chapter is to correct the formula of [74] when k is unbounded, and to provide an analytical approach when k is a fixed constant. The latter approach is a first-order approximation but applies to any probability distribution of errors.

While it is very natural and interesting to consider the latency of error detection, the model of [74] suffers from an important limitation: it is not clear how one can determine when the error has indeed occurred, and hence to identify the last valid checkpoint, unless some verification system is enforced. Another major contribution of this Chapter is to introduce a model coupling verification and checkpointing, and to analytically determine the best balance between checkpoints and verifications so as to optimize platform throughput.

The rest of the Chapter is organized as follows. First we revisit the multiple checkpointing model of [74] in Section 5.2; we tackle both the case where all checkpoints are kept, and the case with at most k checkpoints. In Section 5.3, we define and analyze a model coupling checkpoints and verifications. Then, we evaluate the various models in Section 5.4, by instantiating the models with realistic parameters derived from future exascale platforms. Finally, we conclude and discuss future research directions in Section 5.5.

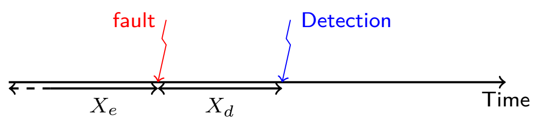


Figure 5.1: Error and detection latency.

5.2 Revisiting the multiple checkpointing model

In this section, we revisit the approach of [74]. We show that their analysis with unbounded memory is incorrect and provide the exact solution (Section 5.2.1). We also extend their approach to deal with the case where a given (constant) number of checkpoints can be simultaneously kept in memory (Section 5.2.2).

5.2.1 Unlimited checkpoint storage

Let C be the time needed for a checkpoint, R the time for recovery, and D the downtime. Although R and C are a function of the size of the memory footprint of the process, D is a constant that represents the unavoidable costs to rejuvenate a process after an error (e.g., stopping the failed process and restoring a new one that will load the checkpoint image). We assume that errors can take place during checkpoint and recovery but not during downtime (otherwise, the downtime could be considered part of the recovery).

Let $\mu_e = \frac{1}{\lambda_e}$ be the mean time between errors. With no error detection latency and no downtime, well-known formulas for the optimal period (useful work plus checkpointing time that minimizes the execution time) are $T_{\text{opt}} \approx \sqrt{2C\mu_e} + C$ (as given by Young [53]) and $T_{\text{opt}} \approx \sqrt{2C(\mu_e + R)} + C$ (as given by Daly [54]). These formulas are first-order approximations and are valid only if $C, R \ll \mu_e$ (in which case they collapse).

With error detection latency, things are more complicated, even with the assumption that one can track the source of the error (and hence identify the last valid checkpoint). Indeed, the amount of rollback will depend upon the sum $X_e + X_d$. For Exponential distributions of X_e and X_d, Lu, Zheng and Chien [74] derive that $T_{\text{opt}} \approx \sqrt{2C(\mu_e + \mu_d)} + C$, where $\mu_d = \frac{1}{\lambda_d}$ is the mean of error detection times. However, although this result may seem intuitive, it is wrong, and we prove that the correct answer is $T_{\text{opt}} \approx \sqrt{2C\mu_e} + C$, even when accounting for the downtime: this first-order approximation is the same as Young's formula. We give an intuitive explanation after the proofs provided in Section 5.2.1. Then in Section 5.2.1, we extend this result to arbitrary laws, but under the additional constraint that $\mu_d + D + R \ll \mu_e$.

Exponential distributions

In this section, we assume that X_e and X_d follow Exponential distributions of mean μ_e and μ_d respectively.

Proposition 10. *The expected time needed to successfully execute a work of size w followed by its checkpoint is*

$$\mathbb{E}(T(w)) = e^{\lambda_e R}\,(D + \mu_e + \mu_d)\,(e^{\lambda_e(w+C)} - 1).$$

Proof. Let $T(w)$ be the time needed for successfully executing a work of duration w. There are two cases: (i) if there is no error during execution and checkpointing, then the time needed is exactly $w+C$; (ii) if there is an error before successfully completing the work and its checkpoint, then some additional delays are incurred. These delays come from three sources: the time spent computing by the processors before the error occurs, the time spent before the error is detected, and the time spent for downtime and recovery. Regardless, once a successful recovery has been completed, there still remain w units of work to execute. Thus, we can write the following recursion:

$$\mathbb{E}(T(w)) = e^{-\lambda_e(w+C)}(w+C) + (1-e^{-\lambda_e(w+C)})\left(\mathbb{E}(T_{lost}) + \mathbb{E}(X_d) + \mathbb{E}(T_{rec}) + \mathbb{E}(T(w))\right). \quad (5.1)$$

Here, T_{lost} denotes the amount of time spent by the processors before the first error, knowing that this error occurs within the next $w+C$ units of time. In other terms, it is the time that is wasted because computation and checkpoint were not both completed before the error occurred. The random variable X_d represents the time needed for error detection, and its expectation is $\mathbb{E}(X_d) = \mu_d = \frac{1}{\lambda_d}$. The last variable T_{rec} represents the amount of time needed by the system to perform a recovery. Equation (5.1) simplifies to:

$$\mathbb{E}(T(w)) = w + C + (e^{\lambda_e(w+C)} - 1)(\mathbb{E}(T_{lost}) + \mu_d + \mathbb{E}(T_{rec})). \quad (5.2)$$

We have

$$\begin{aligned}\mathbb{E}(T_{lost}) &= \int_0^\infty x\mathbb{P}(X = x | X < w+C)dx \\ &= \frac{1}{\mathbb{P}(X < w+C)} \int_0^{w+C} x\lambda_e e^{-\lambda_e x}dx,\end{aligned}$$

and $\mathbb{P}(X < w+C) = 1 - e^{-\lambda_e(w+C)}$.
Integrating by parts, we derive that

$$\mathbb{E}(T_{lost}) = \frac{1}{\lambda_e} - \frac{w+C}{e^{\lambda_e(w+C)} - 1}. \quad (5.3)$$

Next, to compute $\mathbb{E}(T_{rec})$, we have a recursive equation quite similar to Equation (5.1) (remember that we assumed that no error can take place during the downtime):

$$\mathbb{E}(T_{rec}) = e^{-\lambda_e R}(D+R) + (1-e^{-\lambda_e R})(\mathbb{E}(R_{lost}) + \mathbb{E}(X_d) + D + \mathbb{E}(T_{rec})).$$

Here, $\mathbb{E}(R_{lost})$ is the expected amount of time lost to executing the recovery before an error happens, knowing that this error occurs within the next R units of time. Replacing $w+C$ by R in Equation (5.3), we obtain

$$\mathbb{E}(R_{lost}) = \frac{1}{\lambda_e} - \frac{R}{e^{\lambda_e R} - 1}.$$

The expression for $\mathbb{E}(T_{rec})$ simplifies to

$$\mathbb{E}(T_{rec}) = De^{\lambda_e R} + (e^{\lambda_e R} - 1)(\mu_e + \mu_d). \quad (5.4)$$

Plugging the values of $\mathbb{E}(T_{lost})$ and $\mathbb{E}(T_{rec})$ into Equation (5.2) leads to the desired value. ■

Proposition 11. *The optimal strategy to execute a work of size W is to divide it into n equal-size chunks, each followed by a checkpoint, where n is equal either to $\max(1, \lfloor n^* \rfloor)$ or to $\lceil n^* \rceil$. The value of n^* is uniquely derived from $y = \frac{\lambda_e W}{n^*} - 1$, where $\mathbb{L}(y) = -e^{-\lambda_e C - 1}$ ($\mathbb{L}$, the Lambert function, defined as $\mathbb{L}(x)e^{\mathbb{L}(x)} = x$). The optimal strategy does not depend on the value of μ_d.*

Proof. Using n chunks of size w_i (with $\sum_{i=1}^n w_i = W$), by linearity of the expectation, we have $\mathbb{E}(T(W)) = K\sum_{i=1}^n (e^{\lambda_e(w_i+C)} - 1)$ where $K = e^{\lambda_e R}(D + \mu_e + \mu_d)$ is a constant. By convexity, the sum is minimum when all the w_is are equal (to $\frac{W}{n}$). Now, $\mathbb{E}(T(W))$ is a convex function of n, hence it admits a unique minimum n^* such that the derivative is zero:

$$e^{\lambda_e(\frac{W}{n^*}+C)}\left(1 - \frac{\lambda_e W}{n^*}\right) = 1. \tag{5.5}$$

Let $y = \frac{\lambda_e W}{n^*} - 1$, we have $ye^y = -e^{-\lambda_e C - 1}$, hence $\mathbb{L}(y) = -e^{-\lambda_e C-1}$. Then, since we need an integer number of chunks, the optimal strategy is to split W into $\max(1, \lfloor n^*\rfloor)$ or $\lceil n^*\rceil$ same-size chunks, whichever leads to the smaller value. As stated, the value of y, hence of n^*, is independent of μ_d. ■

Proposition 12. *A first-order approximation for the optimal checkpointing period (that minimizes total execution time) is $T_{opt} \approx \sqrt{2C\mu_e} + C$. This value is identical to Young's formula, and does not depend on the value of μ_d.*

Proof. We use Proposition 11 and Taylor expansions when $z = y + 1 = \frac{\lambda_e W}{n^*}$ is small: from $ye^y = -e^{-\lambda_e C-1}$, we derive $(z-1)e^z = -e^{-\lambda_e C}$. We have $(z-1)e^z \approx \frac{z^2}{2} - 1$, and $-e^{-\lambda_e C} \approx -1 + \lambda_e C$, hence $z^2 \approx 2\lambda_e C$. The period is

$$T_{\text{opt}} = \frac{W}{n^*} + C = \frac{z}{\lambda_e} + C \approx \sqrt{2C\mu_e} + C.$$

■

An intuitive explanation of the result is the following: error detection latency is paid for every error, and can be viewed as an additional downtime, which has no impact on the optimal period.

Arbitrary distributions

Here we extend the previous result to arbitrary distribution laws for X_e and X_d (of mean μ_e and μ_d respectively):

Proposition 13. *When $C \ll \mu_e$ and $\mu_d + D + R \ll \mu_e$, a first-order approximation for the optimal checkpointing period is $T_{opt} \approx \sqrt{2C\mu_e} + C$.*

Proof. Let $\mathcal{T}_{\text{base}}$ be the base time of the application without any overhead due to resilience techniques. First, assume a fault-free execution of the application: every period of length T, only $W = T - C$ units of work are executed, hence the time $\mathcal{T}_{\text{ff}}$ for a fault-free execution is $\mathcal{T}_{\text{ff}} = \frac{T}{W}\mathcal{T}_{\text{base}}$. Now, let $\mathcal{T}_{\text{final}}$ denote the expectation of the execution time with errors taken into account. In average, errors occur every μ_e time-units, and for each of them we lose $\mathcal{F}$ time-units, so there are $\frac{\mathcal{T}_{\text{final}}}{\mu_e}$ errors during the execution. Hence we derive that

$$\mathcal{T}_{\text{final}} = \mathcal{T}_{\text{ff}} + \frac{\mathcal{T}_{\text{final}}}{\mu_e}\mathcal{F}, \tag{5.6}$$

which we rewrite as

$$(1-\textsc{Waste})\mathcal{T}_{\text{final}} = \mathcal{T}_{\text{base}},$$
$$\text{with } \textsc{Waste} = 1 - \left(1 - \frac{\mathcal{F}}{\mu_e}\right)\left(1 - \frac{C}{T}\right). \tag{5.7}$$

The waste is the fraction of time where nodes do not perform useful computations. Minimizing execution time is equivalent to minimizing the waste. In Equation (5.7), we identify the two sources of overhead: (i) the term $\textsc{Waste}_{\text{ff}} = \frac{C}{T}$, which is the waste due to checkpointing in a fault-free execution, by construction of the algorithm; and (ii) the term $\textsc{Waste}_{\text{fail}} = \frac{\mathcal{F}}{\mu_e}$, which is the waste due to errors striking during execution. With these notations, we have

$$\textsc{Waste} = \textsc{Waste}_{\text{fail}} + \textsc{Waste}_{\text{ff}} - \textsc{Waste}_{\text{fail}}\textsc{Waste}_{\text{ff}}. \tag{5.8}$$

As a sanity check, this is the same equation as Equation 1.13. There remains to determine the (expected) value of $\mathcal{F}$. Assuming at most one error per period, we lose $\mathcal{F} = \frac{T}{2} + \mu_d + D + R$ per error: $\frac{T}{2}$ for the average work lost before the error occurs, μ_d for detecting the error, and $D + R$ for downtime and recovery. Note that the assumption is valid only if $\mu_d + D + R \ll \mu_e$ and $T \ll \mu_e$. Plugging back this value into Equation (5.8), we obtain

$$\textsc{Waste}(T) = \frac{T}{2\mu_e} + \frac{C(1 - \frac{D+R+\mu_d}{\mu_e})}{T} + \frac{D + R + \mu_d - \frac{C}{2}}{\mu_e} \tag{5.9}$$

which is minimal for

$$T_{\text{opt}} = \sqrt{2C(\mu_e - D - R - \mu_d)}. \tag{5.10}$$

We point out that this approach based on the waste leads to a different approximation formula for the optimal period, but $T_{\text{opt}} = \sqrt{2C(\mu_e - D - R - \mu_d)} \approx \sqrt{2C\mu_e} \approx \sqrt{2C\mu_e} + C$ up to second-order terms, when μ_e is large in front of the other parameters, including μ_d. For example, this approach does not allow us to handle the case $\mu_d = \mu_e$; in such a case, the optimal period is known only for Exponential distributions, and is independent of μ_d, as proven by Proposition 11. ■

To summarize, the exact value of the optimal period is only known for Exponential distributions and is provided by Proposition 11, while Young's formula can be used as a first-order approximation for any other distributions. Indeed, the optimal period is a trade-off between the overhead due to checkpointing ($\frac{C}{T}$) and the expected time lost per error ($\frac{T}{2\mu_e}$ plus some constant). Up to second-order terms, the waste is minimum when both factors are equal, which leads to Young's formula, and which remains valid regardless of error detection latencies.

5.2.2 Saving only k checkpoints

Lu, Zheng and Chien [74] propose a set of simulations to assess the overhead induced when keeping only the last k checkpoints (because of storage limitations). In the following, we derive an analytical approach to numerically solve the problem. The main difficulty is that when error detection latency is too large, it is impossible to recover from a valid checkpoint, and one must resume the execution from scratch. We consider this scenario as an *irrecoverable failure*, and we aim at guaranteeing that the risk of irrecoverable failure remains under a user-given threshold.

Assume that a job of total size W is partitioned into n chunks. What is the risk of irrecoverable failure during the execution of one chunk of size $\frac{W}{n}$ followed by its checkpoint? Let $T = \frac{W}{n} + C$ be the length of the period. Intuitively, the longer the period, the smaller the probability that an error that has just been detected took place more than k periods ago, thereby leading to an irrecoverable failure because the last valid checkpoint is not one of the k most recent ones.

Formally, there is an irrecoverable failure if: (i) there is an error detected during the period (probability $\mathbb{P}_{\text{fail}}$), and (ii) the sum of T_{lost}, the time elapsed since the last checkpoint, and of

X_d, the error detection latency, exceeds kT (probability $\mathbb{P}_{\text{lat}}$). The value of $\mathbb{P}_{\text{fail}} = \mathbb{P}(X_e \leq T)$ is easy to compute from the error distribution law. For instance with an Exponential law, $\mathbb{P}_{\text{fail}} = 1 - e^{-\lambda_e T}$. As for $\mathbb{P}_{\text{lat}}$, we use an upper bound: $\mathbb{P}_{\text{lat}} = \mathbb{P}(T_{lost} + X_d \geq kT) \leq \mathbb{P}(T + X_d \geq kT) = \mathbb{P}(X_d \geq (k-1)T)$. The latter value is easy to compute from the error distribution law. For instance with an Exponential law, $\mathbb{P}_{\text{lat}} = e^{-\lambda_d (k-1)T}$. Of course, if there is an error and the error detection latency does not exceed kT (probability (1-$\mathbb{P}_{\text{lat}}$)), we have to restart execution and face the same risk as before. Therefore, the probability of irrecoverable failure $\mathbb{P}_{\text{irrec}}$ can be recursively evaluated as $\mathbb{P}_{\text{irrec}} = \mathbb{P}_{\text{fail}}(\mathbb{P}_{\text{lat}} + (1 - \mathbb{P}_{\text{lat}})\mathbb{P}_{\text{irrec}})$, hence $\mathbb{P}_{\text{irrec}} = \frac{\mathbb{P}_{\text{fail}}\mathbb{P}_{\text{lat}}}{1-\mathbb{P}_{\text{fail}}(1-\mathbb{P}_{\text{lat}})}$. Now that we have computed $\mathbb{P}_{\text{irrec}}$, the probability of irrecoverable failure for a single chunk, we can compute the probability of irrecoverable failure for n chunks as $\mathbb{P}_{\text{risk}} = 1 - (1 - \mathbb{P}_{\text{irrec}})^n$. In full rigor, these expressions for $\mathbb{P}_{\text{irrec}}$ and $\mathbb{P}_{\text{risk}}$ are valid only for Exponential distributions, because of the memoryless property, but they are a good approximation for arbitrary laws. Given a prescribed risk threshold ε, solving numerically the equation $\mathbb{P}_{\text{risk}} \leq \varepsilon$ leads to a lower bound T_{min} on T. Let T_{opt} be the optimal period given in Theorem 12 for an unbounded number of saved checkpoints. The best strategy is then to use the period $\max(T_{\text{min}}, T_{\text{opt}})$ to minimize the waste while enforcing a risk below threshold.

In case of irrecoverable failure, we have to resume execution from the very beginning. The number of re-executions due to consecutive irrecoverable failures follows a geometric law of parameter $1 - \mathbb{P}_{\text{risk}}$, so that the expected number of executions until success is $\frac{1}{1-\mathbb{P}_{\text{risk}}}$. We refer to Section 5.4.1 for an example of how to instantiate this model to compute the best period with a fixed number of checkpoints, under a prescribed risk threshold.

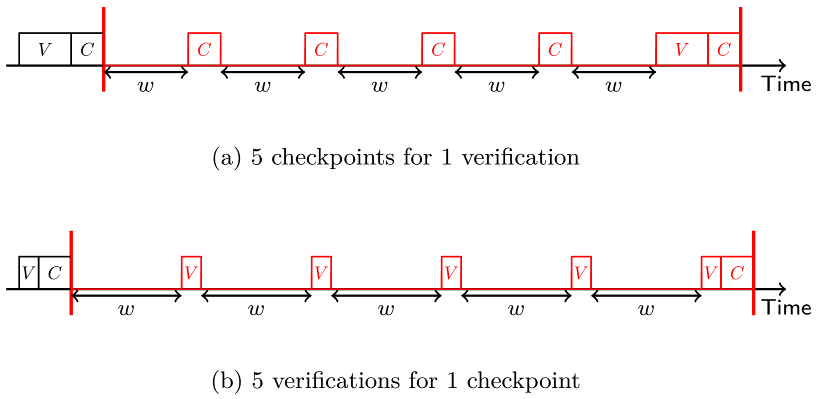
5.3 Coupling verification and checkpointing

In this section, we move to a more realistic model where silent errors are detected only when some verification mechanism (checksum, error correcting code, coherence tests, etc.) is executed. Our approach is agnostic of the nature of this verification mechanism. We aim at solving the following optimization problem: given the cost of checkpointing C, downtime D, recovery R, and verification V, what is the optimal strategy to minimize the expected waste as a function of the mean time between errors μ_e? Depending upon the relative costs of checkpointing and verifying, we may have more checkpoints than verifications, or the other way round. In both cases, we target a periodic pattern that repeats over time.

Consider first the scenario where the cost of a checkpoint is smaller than the cost of a verification: then the periodic pattern will include k checkpoints and 1 verification, where k is some parameter to determine. Figure 5.2(a) provides an illustration with $k = 5$. We assume that the verification is directly followed by the last checkpoint in the pattern, so as to save results just after they have been verified (and before they get corrupted). In this scenario, the objective is to determine the value of k that leads to the minimum platform waste. This problem is addressed in Section 5.3.1.

Because our approach is agnostic of the cost of the verification, we also envision scenarios where the cost of a checkpoint is higher than the cost of a verification. In such a framework, the periodic pattern will include k verifications and 1 checkpoint, where k is some parameter to determine. See Figure 5.2(b) for an illustration with $k = 5$. Again, the objective is to determine the value of k that leads to the minimum platform waste. This problem is addressed in Section 5.3.2.

We point out that combining verification and checkpointing guarantees that no irrecoverable failure will kill the application: the last checkpoint of any period pattern is always correct, because a verification always takes place right before this checkpoint is taken. If that verification reveals an error, we roll back until reaching a correct verification point, maybe up to the end of the previous pattern, but never further back, and re-execute the work. The amount of roll-back and re-execution depends upon the shape of the pattern, and we show how to compute it in Sections 5.3.1 and 5.3.2 below.



(a) 5 checkpoints for 1 verification

(b) 5 verifications for 1 checkpoint

Figure 5.2: Periodic pattern.

5.3.1 With k checkpoints and 1 verification

We use the same approach as in the proof of Proposition 13 and compute a first-order approximation of the waste (see Equations (5.7) and (5.8)). We compute the two sources of overhead: (i) WASTE_{ff}, the waste incurred in a fault-free execution, by construction of the algorithm, and (ii) $\text{WASTE}_{\text{fail}}$, the waste due to errors striking during execution.

Let $\mathbb{S} = kw + kC + V$ be the length of the periodic pattern. We easily derive that $\text{WASTE}_{\text{ff}} = \frac{kC+V}{\mathbb{S}}$. As for $\text{WASTE}_{\text{fail}}$, we still have $\text{WASTE}_{\text{fail}} = \frac{D+\mathbb{E}(T_{lost})}{\mu_e}$. However, in this context, the time lost because of the error depends upon the location of this error within the periodic pattern, so we compute averaged values as follows. Recall (see Figure 5.2(a)) that checkpoint k is the one preceded by a verification. Here is the analysis when an error is detected during the verification that takes place in the pattern:

— If the error took place in the (last) segment k: we recover from checkpoint $k-1$, and verify it; we get a correct result because the error took place later on. Then we re-execute the last piece of work and redo the verification. The time that has been lost is $T_{lost}(k) = R + V + w + V$. (We assume that there is at most one error per pattern.)

— If the error took place in segment i, $2 \leq i \leq k-1$: we recover from checkpoint $k-1$, verify it, get a wrong result; we iterate, going back up to checkpoint $i-1$, verify it, and get a correct result because the error took place later on. Then we re-execute $k-i+1$ pieces of work and $k-i$ checkpoints, together with the last verification. We get $T_{lost}(i) = (k-i+1)(R+V+w) + (k-i)C + V$.

— If the error took place in (first) segment 1: this is almost the same as above, except that the first recovery at the beginning of the pattern need not be verified, because the verification was made just before the corresponding checkpoint at the end of the previous pattern. We have the same formula with $i = 1$ but with one fewer verification: $T_{lost}(1) = k(R+w) + (k-1)(C+V) + V$.

Therefore, the formula for $\textsc{Waste}_{\text{fail}}$ writes

$$\textsc{Waste}_{\text{fail}} = \frac{D + \frac{1}{k}\sum_{i=1}^{k} T_{lost}(i)}{\mu_e}, \tag{5.11}$$

and (after some manipulation using a computer algebra system) the formula simplifies to

$$\textsc{Waste}_{\text{fail}} = \frac{1}{2k\mu_e}((R+V)k^2+(2D+R+2V+\mathbb{S}-2C)k+\mathbb{S}-3V) \tag{5.12}$$

Using $\textsc{Waste}_{\text{ff}} = \frac{kC+V}{\mathbb{S}}$ and Equation (5.8), we compute the total waste and derive that $\textsc{Waste} = a\mathbb{S}+b+\frac{c}{\mathbb{S}}$, where a, b, and c are some constants. The optimal value of $\mathbb{S}$ is $\mathbb{S}_{opt} = \sqrt{\frac{c}{a}}$, provided that this value is at least $kC+V$. We point out that this formula only is a first-order approximation. We have assumed a single error per pattern. We have also assumed that errors did not occur during checkpoints following verifications. Now, once we have found $\textsc{Waste}(\mathbb{S}_{opt})$, the value of the waste obtained for the optimal period $\mathbb{S}_{opt}$, we can minimize this quantity as a function of k, and numerically derive the optimal value k_{opt} that provides the best value (and hence the best platform usage).

Due to lack of space, computational details are available in [93], which is a Maple sheet that we have to instantiate the model. This Maple sheet is publicly available for users to experiment with their own parameters. We provide two example scenarios to illustrate the model in Section 5.4.3.

Finally, note that in order to minimize the waste, one could do a binary search in order to find the last checkpoint before the fault. Then we can upper-bound $T_{lost}(i)$ by $(k-i+1)w + \log(k)(R+V)+(k-i)C+V$, and Equation (5.12) becomes $\textsc{Waste}_{\text{fail}} = \frac{1}{2k\mu_e}((R+V)2k\log(k)+(2D+R+2V+\mathbb{S}-2C)k+\mathbb{S}-3V)$.

5.3.2 With k verifications and 1 checkpoint

We use a similar line of reasoning for this scenario and compute a first-order approximation of the waste for the case with k verifications and 1 checkpoint per pattern. The length of the periodic pattern is now $\mathbb{S} = kw + kV + C$. As before, for $1 \leq i \leq k$, let segment i denote the period of work before verification i, and assume (see Figure 5.2(b)) that verification k is preceded by a checkpoint. The analysis is somewhat simpler here.

If an error takes place in segment i, $1 \leq i \leq k$, we detect the error during verification i, we recover from the last checkpoint, and redo the first i segments and verifications: therefore $T_{lost}(i) = R + i(V+w)$. The formula for $\textsc{Waste}_{\text{fail}}$ is the same as in Equation (5.11) and (after some manipulation) we derive

$$\textsc{Waste}_{\text{fail}} = \frac{1}{2\mu_e}\left(D + R + \frac{k+1}{2k}(\mathbb{S}-C)\right). \tag{5.13}$$

Using $\textsc{Waste}_{\text{ff}} = \frac{kV+C}{\mathbb{S}}$ and Equation (5.8), we proceed just as in Section 5.3.1 to compute the optimal value $\mathbb{S}_{opt}$ of the periodic pattern, and then the optimal value k_{opt} that minimizes the waste. Details are available within the Maple sheet [93].

5.4 Evaluation

This section provides some examples for instantiating the various models. We aimed at choosing realistic parameters in the context of future exascale platforms, but we had to restrict

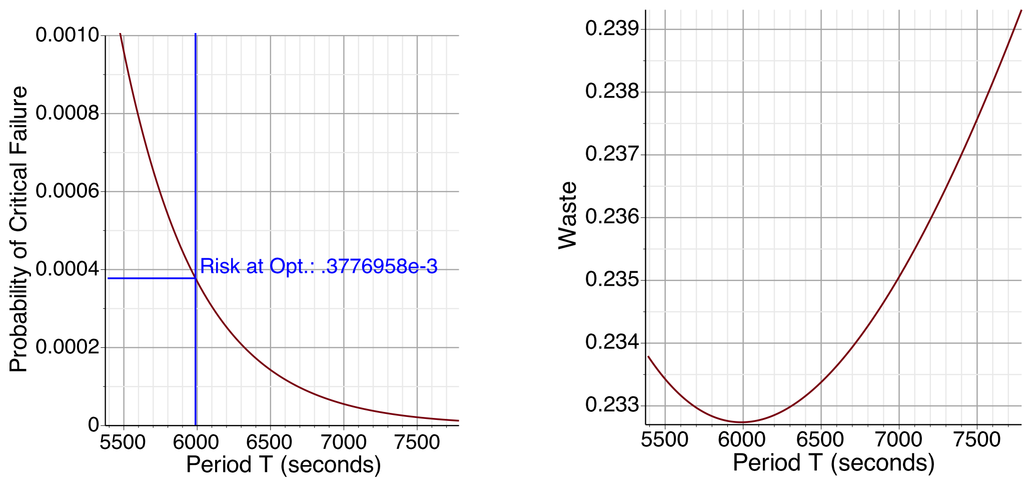


Figure 5.3: Risk of irrecoverable failure as a function of the checkpointing period, and corresponding waste. ($k = 3$, $\lambda_e = \frac{10^5}{100y}$, $\lambda_d = 30\lambda_e$, $w = 10d$, $C = R = 600s$, and $D = 0s$.)

to a limited set of scenarios, which do not intend to cover the whole spectrum of possible parameters. The Maple sheet [93] is available to explore other scenarios.

5.4.1 Best period with k checkpoints under a given risk threshold

We first evaluate $\mathbb{P}_{\text{risk}}$, the risk of irrecoverable failure, as defined in Section 5.2.2. Figures 5.3 and 5.4 present, for different scenarios, the probability $\mathbb{P}_{\text{risk}}$ as a function of the checkpointing period T on the left. On the right, the figures present the corresponding waste with k checkpoints and in the absence of irrecoverable failures. This waste can be computed following the same reasoning as in Equation (5.9). For each figure, the left diagram represents the risk implied by a given period T, showing the value T_{opt} of the optimal checkpoint interval (optimal with respect to waste minimization and in the absence of irrecoverable failures, see Equation (5.10)) as a blue vertical line. The right diagram on the figure represents the corresponding waste, highlighting the trade-off between an increased irrecoverable-failure-free waste and a reduced risk. As stated in Section 5.2.2, it does not make sense to select a value for T lower than T_{opt}, since the waste would be increased, for an increased risk.

Figure 5.3 considers a machine consisting of 10^5 components, and a component MTBF of 100 years. This component MTBF corresponds to the optimistic assumption on the reliability of computers made in the literature [91, 92]. The platform MTBF μ_e is thus $100 \times 365 \times 24/100,000 \approx 8.76$ hours. The times to checkpoint and recover (10 min) correspond to reasonable mean values for systems at this size [61, 69]. At this scale, process rejuvenation is small, and we set the downtime to 0s. For these average values to have a meaning, we consider a run that is long enough (10 days of work), and in order to illustrate the trade-off, we take a rather low (but reasonable) value $k = 3$ of intervals, and a mean time error detection μ_d significantly smaller (30 times) than the MTBF μ_e itself.

With these parameters, T_{opt} is around 100 minutes, and the risk of irrecoverable failure at this checkpoint interval can be evaluated at $1/2617 \approx 38 \cdot 10^{-5}$, inducing an irrecoverable-failure-free waste of 23.45%. To reduce the risk to 10^{-4}, a T_{min} of 8000 seconds is sufficient, increasing

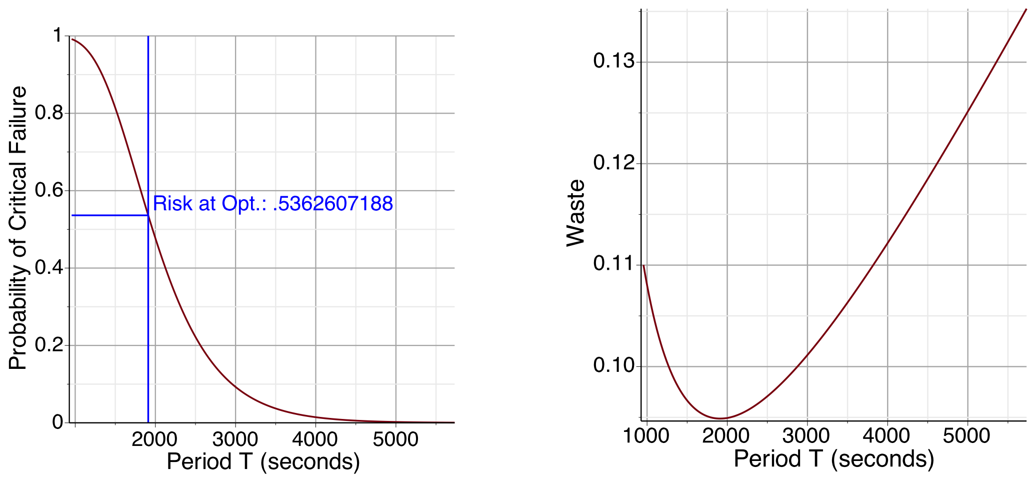


Figure 5.4: Risk of irrecoverable failure as a function of the checkpointing period, and corresponding waste. ($k = 3$, $\lambda_e = \frac{10^5}{100y}, \lambda_d = 30\lambda_e, w = 10d, C = R = 60s$, and $D = 0s$.)

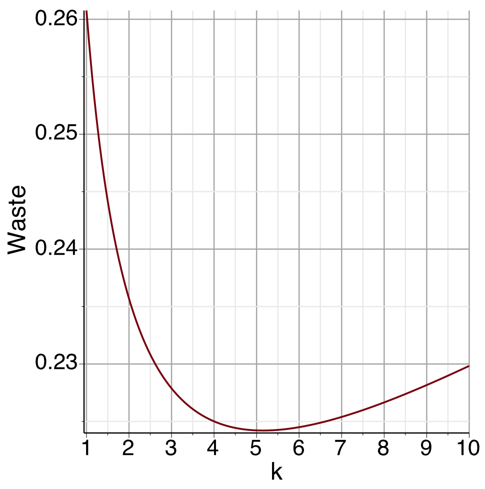
the waste by only 0.6%. In this case, the benefit of fixing the period to $\max(T_{\text{opt}}, T_{\text{min}})$ is obvious. Naturally, keeping a bigger amount of checkpoints (increasing k) would also reduce the risk, at constant waste, if memory can be afforded.

We also consider in Figure 5.4 a more optimistic scenario where the checkpointing technology and availability of resources is increased by a factor 10: the time to checkpoint, recover, and allocate new computing resources is divided by 10 compared to the previous scenario. Other parameters are kept similar. One can observe that T_{opt} is largely reduced (down to less than 35 minutes between checkpoints), as well as the optimal irrecoverable-failure-free waste (9.55%). This is unsurprising, and mostly due to the reduction of failure-free waste implied by the reduction of checkpointing time. But because the period between checkpoints becomes smaller, while the latency to detect an error is unchanged (μ_d is still 30 times smaller than μ_e), the risk that an error happens at the interval i but is detected after interval $i+k$ is increased. Thus, the risk climbs to 1/2, an unacceptable value. To reduce the risk to 10^{-4} as previously, it becomes necessary to consider a T_{min} of 6650 seconds, which implies an irrecoverable-failure-free waste of 15%, significantly higher than the optimal one, which is below 10%, but still much lower than the 24% when checkpoint and availability costs are 10 times higher.

5.4.2 Periodic pattern with k verifications and 1 checkpoint

We now focus on the waste induced by the different ways of coupling periodic verification and checkpointing. We first consider the case of a periodic pattern with more verifications than checkpoints: every k verifications of the current state of the application, a checkpoint is taken. The duration of the work interval $\mathbb{S}$, between two verifications in this case, is optimized to minimize the waste. We consider two scenarios. For each scenario, we represent two diagrams: the left diagram shows the waste as a function of k for a given verification cost V, and the right diagram shows the waste as a function of k and V using a 3D surface representation.

In the first scenario, we consider the same setup as above in Section 5.4.1. The waste is computed in its general form, so we do not need to define the duration of the work. As



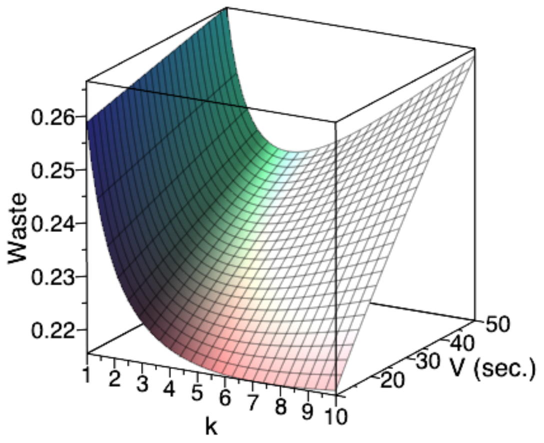


Figure 5.5: Case with k verifications, and one checkpoint per periodic pattern. Waste as function of k, and potentially of V, using the optimal period. ($V = 20s, C = R = 600s, D = 0s$, and $\mu = \frac{10y}{10^5}$.)

represented in Figure 5.5, for a given verification cost, the waste can be optimized by making more than one verifications. When $k > 1$, there are intermediate verifications that can enable to detect an error before a periodic pattern (of length $\mathbb{S}$) is completed, hence, that can reduce the time lost due to an error. However, introducing too many verifications induces an overhead that eventually dominates the waste. The 3D surface shows that the waste reduction is significant when increasing the number of verifications, until the optimal number is reached. Then, the waste starts to increase again slowly. Intuitively, the lower the cost for V, the higher the optimal value for k.

When considering the second scenario (Figure 5.6), with an improved checkpointing and availability setup, the same conclusions can be reached, with an absolute value of the waste greatly diminished. Since forced verifications allow to detect the occurrence of errors at a controllable rate (depending on $\mathbb{S}$ and k), the risk of non-recoverable errors is nonexistent in this case, and the waste can be greatly diminished, with very few checkpoints taken and kept during the execution.

5.4.3 Periodic pattern with k checkpoints and 1 verification

The last set of experiments considers the opposite case of periodic patterns: checkpoints are taken more often than verifications. Every k checkpoints, a verification of the data consistency is done. Intuitively, this could be useful if the cost of verification is large compared to the cost of checkpointing itself. In that case, when rolling back after an error is discovered, each checkpoint that was not validated before is validated at rollback time, potentially invalidating up to $k - 1$ checkpoints.

Because this pattern has potential only when the cost of checkpoint is much lower than the cost of verification, we considered the case of a greatly improved checkpoint / availability setup: the checkpoint and recovery times are only 6 seconds in Figure 5.7. One can observe that in this extreme case, it can still make sense to consider multiple checkpoints between two verifications (when $V = 100$ seconds, a verification is done only every 3 checkpoints optimally); however the 3D surface demonstrates that the waste is still dominated by the cost of the verification, and

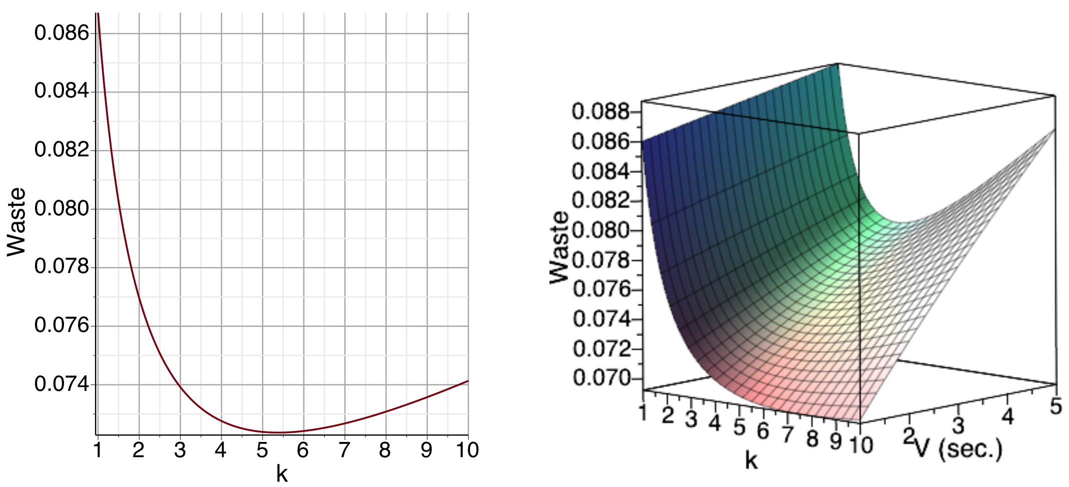


Figure 5.6: Case with k verifications, and one checkpoint per periodic pattern. Waste as function of k, and potentially of V, using the optimal period. ($V = 2s, C = R = 60s, D = 0s$, and $\mu = \frac{10y}{10^5}$.)

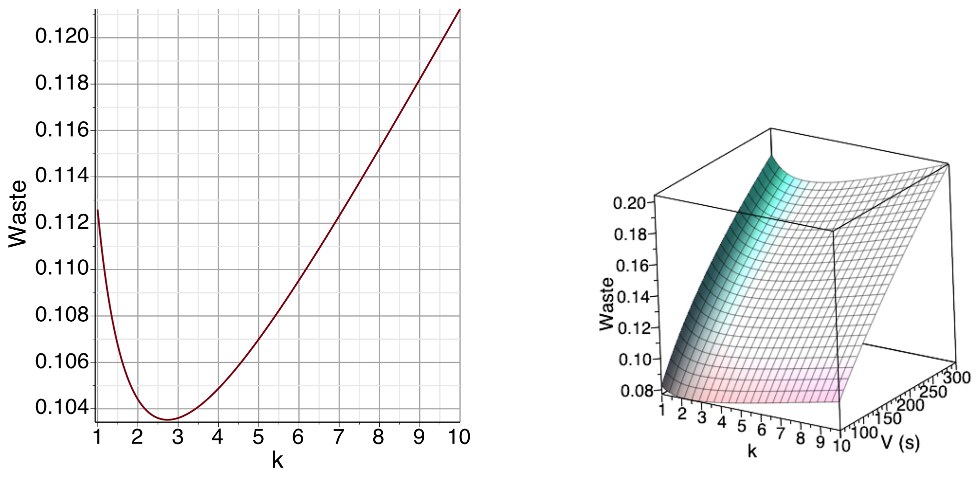


Figure 5.7: Case with k checkpoints, and one verification per periodic pattern. Waste as function of k, and potentially of V, using the optimal period. ($V = 100s, C = R = 6s, D = 0s$, and $\mu = \frac{10y}{10^5}$.)

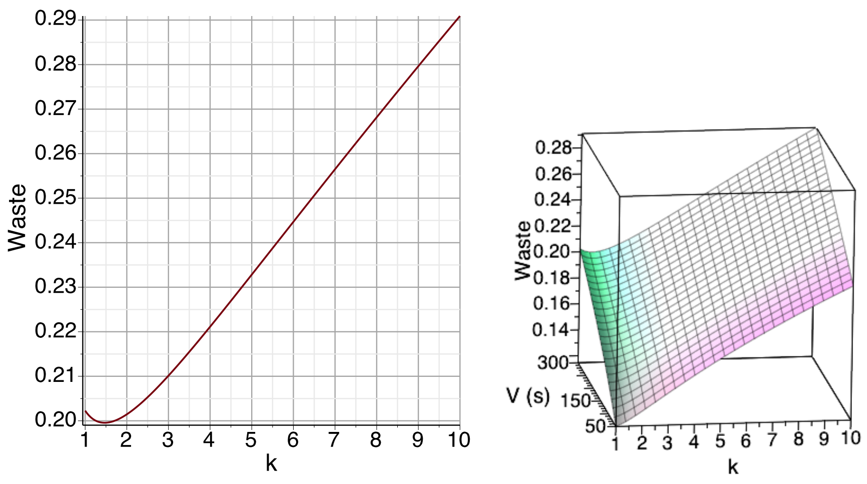


Figure 5.8: Case with k checkpoints, and one verification per periodic pattern. Waste as function of k, and potentially of V, using the optimal period. ($V = 300s, C = R = 60s, D = 0s$, and $\mu = \frac{10y}{10^5}$.)

little improvement can be achieved by taking the optimal value for k. The cost of verification must be incurred when rolling back, and this shows on the overall performance if the verification is costly.

This is illustrated even more clearly with Figure 5.8, where the checkpoint costs and machine availability are set to the second scenario of Sections 5.4.1 and 5.4.2. As soon as the checkpoint cost is not negligible compared to the verification cost (only 5 times smaller in this case), it is more efficient to validate every other checkpoint than to validate only after $k > 2$ checkpoints. The 3D surface shows that this holds true for rather large values of V.

All the rollback / recovery techniques that we have evaluated above, using various parameters for the different costs, and stressing the different approaches to their limits, expose a waste that remains, in the vast majority of the cases, largely below 66%. This is noticeable, because the traditional hardware based technique, which relies on triple modular redundancy and voting [73], mechanically presents a waste that is at least equal to 66% (two-thirds of resources are wasted, and neglecting the cost of voting).

5.5 Conclusion

In this Chapter, we revisit traditional checkpointing and rollback recovery strategies. Rather than considering fail-stop failures, we focus on silent data corruption errors. Such latent errors cannot be neglected anymore in High Performance Computing, in particular in sensitive and high precision simulations. The core difference with fail-stop failures is that error detection is not immediate.

We discuss and analyze two models. In the first model, errors are detected after some delay following a probability distribution (typically, an Exponential distribution). We compute the optimal checkpointing period in order to minimize the waste when all checkpoints can be kept in memory, and we show that this period does not depend on the distribution of detection times. In practice, only a few checkpoints can be kept in memory, and hence it may happen that an error was detected after the last correct checkpoint was removed from storage. We derive a minimum value of the period to guarantee, within a risk threshold, that at least one valid checkpoint remains when a latent error is detected.

A more realistic model assumes that errors are detected through some verification mechanism. Periodically, one checks whether the current status is meaningful or not, and then eventually detects a latent error. We discuss both the case where the periodic pattern includes k checkpoints for one verification (large cost of verification), and the opposite case with k verifications for one checkpoint (inexpensive cost for verification). We express a formula for the waste in both cases, and, from these formulas, we derive the optimal period.

The various models are instantiated with realistic parameters, and the evaluation results clearly corroborate the theoretical analysis. For the first model, with detection times, the tradeoff between waste and risk of irrecoverable error clearly appears, hence showing that a period larger than the one minimizing the irrecoverable-failure-free waste should often be chosen to achieve an acceptable risk. The advantage of the second model is that there are no irrecoverable failures (within each period, there is a verification followed by a checkpoint, hence ensuring a valid checkpoint). We compute the optimal pattern of checkpoints and verifications per period, as a function of their respective cost, to minimize the waste. The pattern with more checkpoints than verification turns out to be usable only when the cost of checkpoint is much lower than the cost of verification, and the conclusion is that it is often more efficient to verify the result every other checkpoint.

Overall, we provide a thorough analysis of checkpointing models for latent errors, both analyzing the models analytically, and evaluating them through different scenarios.

A future research direction would be to study more general scenarios of multiple checkpointing, for instance by keeping not the consecutive k last checkpoints in the first model, but rather some older checkpoints to decrease the risk. In the second model, more verification/checkpoint combinations could be studied, while we focused on cases with an integer number of checkpoints per verification (or the converse).

Part II

Cost-Optimal Execution of Boolean Trees

Chapter 6

Related Work

The problem of computing the truth value of a Boolean query tree while incurring the minimum cost is known as Probabilistic AND-OR Tree Resolution (PAOTR) and has been studied extensively in the literature.

Several works in literature assume that each data stream occurs in at most one leaf of the query tree. This assumption is termed *read-once* queries therein. In this part we study the more general queries, which we term *shared* queries, in which a stream can occur in multiple leaves of the tree.

For *read-once* queries the complexity of the PAOTR problem is unknown for general AND-OR trees. Smith et al. [104] propose a simple $O(n \log n)$ greedy algorithm (n is the number of leaves in the query tree) that produces an optimal leaf evaluation order for AND trees (i.e., single-level trees with an AND operator at the root node). Greiner et al. [105] survey known theoretical results and present several new results. They consider a depth-first approach that recursively replaces rooted subtrees with a single equivalent single node. They show that this approach can be arbitrarily sub-optimal for trees with 3 levels or more. For DNF trees (i.e., collections of AND trees whose roots are the children of a single OR node), they show that this approach is dominant, meaning that there is always one optimal strategy that corresponds to a depth-first traversal. They proposed a $O(n \log n)$ depth-first traversal of the tree that reuses the algorithm in [104] to order leaves within each AND, which produces an optimal evaluation order for any DNF tree.

Kaplan et al. [111] have studied a problem closely related to the PAOTR problem for the *read-once* queries. They have studied an extension of the standard learning models to settings where observing the value of an attribute has an associated cost. Their goal is to design a strategy to decide what attribute of x to observe next so as to minimize the expected evaluation cost of a function f(x). The authors present approximation algorithms to solve this optimization problem. They evaluate a given function f on input x by specifying a decision tree of the attributes. This decision tree is used to decide what attributes of x to observe next, given the outcome of the attributes that they observe. The costs of evaluating an input x using a tree is the sum of the costs of the attributes along the path that x follows in the tree.

Shared queries, the focus of this work, are important in practice and have been introduced and investigated by Lim et al. [100]. In that work the authors do not give theoretical results, but instead develop heuristics to determine an order of operator evaluation that hopefully leads to low data acquisition costs. To the best of our knowledge, the complexity of the PAOTR problem for *shared* queries has never been addressed in the literature, likely because re-using stream data across leaves dramatically complicates the problem. When picking a leaf evaluation order, interdependences between the leaves must be taken into account. In fact, even when given a leaf evaluation order, computing the expected query cost is intricate while this same computation is trivial for *read-once* queries.

Several problems studied in the literature are closely related to the PAOTR problem and fall in the area of system testing [97], and in particular the evaluation of priced functions [109] and the Discrete Function Evaluation Problem (DFEP) [106].

Charikar et al. [109] and Cicalese et al. [110] study the evaluation of "priced functions." In this context an algorithm seeks to read a subset of the function's inputs so that the function's output can be determined with minimum price. When the function is an AND/OR tree there exist algorithms that achieve the best possible competitive ratio in pseudo-polynomial [109] and polynomial [110] time. In this context, no knowledge of the Boolean variables (which correspond to our predicates) is assumed. In the context of our work this would correspond to all predicates having the same probability of success of $\frac{1}{2}$. However, ignoring probabilities of success can lead to solutions that are no better than L-competitive for functions with L inputs, even for a single AND tree [1].

Cicalese et al. [106] have studied the Discrete Function Evaluation Problem (DFEP). An instance of DFEP is defined by a set S of objects, a partition C of these objects into classes (which represent the values taken by the function), a probability distribution p on S, a set T of tests, and a cost function assigning a cost to each test. The goal of DFEP is to design a testing procedure that uses tests from T to identify the class that includes the unknown object, while minimizing the expectation of the testing cost (or the worst-case testing cost). DFEP has also been studied under the names of the Equivalence Class Determination problem [107] and of the Group Identification problem [108].

Evaluation of AND/OR trees is a special case of DFEP where the set of objects is the set of the possible instantiations of the vector of predicates, where the probability distribution is the probability of occurrences of the different instantiations, where there are only two classes corresponding to instantiations leading to the AND/OR tree evaluating to TRUE or FALSE, and where tests correspond to the evaluation of predicates. There exist approximation algorithms for the minimization of the expectation of the testing cost with ratio $O(\log(|S|))$ [106] or $O\left(\log\left(\frac{1}{p_{min}}\right)\right)$ [107, 108], where p_{min} is the minimum probability of an object ($p_{min} = \min_{s \in S} p(s)$). For AND/OR trees, $|S| = 2^L$ and $p_{min} \leq \left(\frac{1}{2}\right)^L$. Therefore, all these approximations algorithms are $O(L)$-approximation algorithms for AND/OR tree evaluation. However, for *single-stream* AND/OR trees without reuse where all costs are identical (what is often called the uniform case), any schedule is a $O(L)$-approximation if the AND/OR tree is neither a tautology nor a false statement: in the best case at least one predicate must be evaluated, and in the worst case all L predicates must be evaluated. Therefore, existing results for DFEP do not lead to efficient solutions for the AND/OR tree evaluation problem.

Moret et al. [112] have studied decision trees and diagrams, also known as sequential evaluation procedures. A decision tree is a model of the evaluation of a discrete function, wherin the value of a variable is determined and the next action is chosen accordingly. The authors

1. Consider an AND node with L leaves of unit cost, $L-1$ having a probability of success of $1-\epsilon$ and the last one having a probability of success of ϵ. If the ϵ-probability leaf is evaluated first the expectation of the cost of the evaluation of the query is $1+\epsilon(1+(1-\epsilon)(1+(1-\epsilon)(...+(1-\epsilon)1))) = 1+\epsilon((1-\epsilon)^0+(1-\epsilon)^1+...+(1-\epsilon)^{L-2}) = 1+(1-(1-\epsilon)^{L-1}) \approx 1+(L-1)\epsilon$ when ϵ tends to zero. When probabilities are ignored, all leaves are absolutely equivalent and a probability-agnostic leaf evaluation order can evaluate the ϵ-probability leaf last, leading to a leaf evaluation order with expected cost equal to: $1+(1-\epsilon)(1+(1-\epsilon)(1+...(1+(1-\epsilon)(1)))) = (1-\epsilon)^0+(1-\epsilon)^1+...+(1-\epsilon)^{L-1} = \frac{1-(1-\epsilon)^L}{1-(1-\epsilon)} \approx L$. Hence the lower bound on the competitive ratio.

present a tutorial survey with recent results and extensions and provide a unified framework of definition and notations for decision trees and diagrams. A decision tree can be regarded as a deterministic algorithm for deciding which variable to test next, based on the previously tested variables and the results of their evaluation, until the function's value can be determined. The evaluation of a discrete function represented as a decision tree starts by ascertaining the value of the variable associated with the root of the tree. It then proceeds by repeating the process on the k^{th} subtree, where k is the value assumed by the root variable, until a leaf is reached; the label of the leaf gives the value of the function. Each variable has an associated testing cost, which measures the expense incurred each time that variable is evaluated, and a storage cost, which measures the expense due to the presence of each test node labeled by that variable.

Chapter 7

Cost-Optimal Execution of Boolean DNF Trees with Shared Streams

7.1 Introduction

An increasing number of applications are being developed or envisioned that continuously process data generated via sensors embedded in or associated with mobile devices. Smartphones are equipped with increasingly sophisticated sensors (e.g., GPS, accelerometer, gyroscope, microphone) that enable near real-time sensing of an individual's activity or environmental context. A smartphone can then perform embedded query processing on the sensor data streams for, e.g., social networking [101], remote health monitoring [102]. Automotive applications running on a smartphone can acquire data from sensors in a vehicle (e.g., engine status, speed, angular speed) as well as from remote databases (e.g., weather, traffic, road works) so as to perform continuous queries that trigger appropriate responses (e.g., alerting the driver that driving conditions are dangerous) [98].

In the above applications there is a cost associated to the acquisition of sensor data. Even moderate data rates can cause commercial smartphone batteries to be depleted in a few hours [103]. In the automotive application scenario, the acquisition of sensor data incurs a cost in terms of bandwidth usage on the sensor network in the vehicle [98]. Consequently, solutions must be developed to reduce the cost of sensor data acquisition when processing continuous queries.

In this work we study the problem of minimizing the (expected) cost of sensor data acquisition when evaluating a query expressed as a tree of conjunctive and disjunctive Boolean operators applied to Boolean *predicates* on the data. Each predicate is computed over data items from different data streams generated periodically by sensors, and has a certain probability of evaluating to true. The evaluation of the query stops as soon as a truth value has been determined, possibly *shortcircuiting* part of the query tree. A "push" model by which sensors continuously transmit data to the device maximizes the amount of acquired data and is thus not practical. Instead, a "pull" model has been proposed [100], by which the query engine carefully chooses the *order* and the *numbers of data items* to acquire from each individual sensor. This choice is based on a-priori knowledge of operator costs and probabilities, which can be inferred based on historical traces obtained for previous query executions. Such intelligent processing is possible thanks to the programming and data filtering capabilities that are emerging on sensor platforms [99, 98].

Three example query trees are shown in Figure 7.1, assuming streams named A, B, and C,

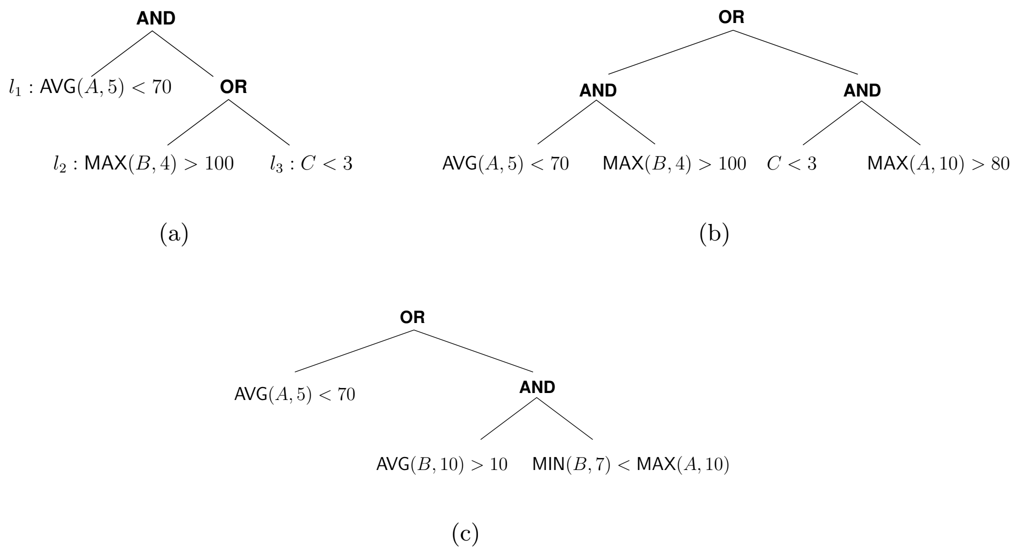


Figure 7.1: Three query tree examples: (a) a *read-once* query; (b) a *shared* query in the *single-stream* case; (c) a *shared* query in the *multi-stream* case.

which produce integer data items. Each leaf corresponds to a Boolean predicate. A predicate may involve no operator, e.g., "$C < 3$" is true if the last item from stream C is strictly lower than 3, or based on an arbitrary operator (in this example MAX, MIN, or AVG) which is applied to a time-window for a stream, e.g., "AVG$(A, 5) < 70$" is true if the average of the last 5 items from A is strictly lower than 70), or multiple operators (e.g., "MIN$(B, 7) <$ MAX$(A, 10)$").

Most results in the literature are for *read-once* queries, i.e., when each data stream occurs in at most one leaf of the query tree. The example query tree in Figure 7.1-(a) is a *read-once* query since no stream occurs in two leaves. In this work we study the more general case, which we term *shared*, in which a stream can occur in multiple leaves. Figure 7.1-(b) shows a *shared* query in which stream A occurs at two leaves. Such queries are easily envisioned in most domains. For instance, in a telehealth example, an alert may be generated either if the heart rate is high and the acceleration is zero or if the heart rate is low and the SPO2 (blood oxygen saturation) is low. *Shared* queries relevant to an automotive application are considered in [98]. We also study the case in which multiple streams can occur in a single predicate. An example is shown in Figure 7.1-(c), in which streams A and B occur in multiple leaves, and together in one leaf. We term the scenarios in Figure 7.1-(a) and 7.1-(b) *single-stream* and the scenario in Figure 7.1-(c) *multi-stream*.

Considering *shared* queries has important algorithmic implications that we explore in this work. The device that processes the query acquires data items from streams and holds each data item in memory until the query has been processed. Each time a leaf of the query must be evaluated, one can then compute the number of data items that must be retrieved from the relevant stream given the time-windows of the operators applied to that stream and the data items from that stream that are already in the device's memory. For example, considering the query in Figure 7.1-(b), assume the predicate "AVG$(A, 5) < 70$" is evaluated first, thus acquiring 5 items from stream A. If later the predicate "MAX$(A, 10) > 80$" needs to be evaluated then only 5 additional items must be acquired.

In this work we study the *shared* queries and make the following contributions:

— For AND query trees:

  — We give a polynomial-time greedy algorithm (which is not as straightforward as the optimal algorithm for *read-once* queries) that is optimal in the *single-stream* case, and show that the problem in NP-complete in the *multi-stream* case.

  — For the *multi-stream* case we propose an extension of the *single-stream* greedy algorithm. This extension is not optimal but computes near-optimal leaf evaluation orders in practice.

— For DNF query trees:

  — We show that the problem is NP-complete in the *single-stream* case (and thus also in the *multi-stream* case).

  — In both the *single-stream* and *multi-stream* case we show that there exists an optimal leaf evaluation order that is depth-first;

  — We develop heuristics that we evaluate in simulation and compare to the optimal solution (computed via an exhaustive search on small instances) and to the *single-stream* heuristic proposed in [100].

Section 7.2 defines the problem and our assumptions formally. Section 7.3 gives a method for computing the expected cost of a leaf evaluation order. We then study the problem for AND trees and DNF trees in Section 7.4 and Section 7.5, respectively, for both the *single-stream* and *multi-stream* cases. Section 7.6 concludes the chapter with a brief summary of our findings.

7.2 Problem statement / Framework

A query is an AND-OR *tree*, i.e., a rooted tree whose non-leaf nodes are AND or OR operators, and whose leaves are labeled with probabilistic Boolean predicates. Each predicate is evaluated over data items generated by data *streams*. The evaluation of each predicate has a known *success probability*, i.e., the probability that the predicate evaluates to TRUE. In practice, the success probability can be estimated based on historical traces obtained from previous query evaluations. As in [105], we assume *independent* predicates, meaning that two predicates at two leaves in a query are statistically independent. Evaluating a predicate incurs has a *cost* determined by the number of data items required to perform the evaluation and a per data item cost for the stream. For instance, the cost of a data item could correspond to the energy cost, in joules, of acquiring one data item based on the communication medium used for the stream and the data item size.

More formally, we consider a set of streams, $\mathcal{S} = \{s_1, \ldots, s_S\}$. Stream s_k has a cost per data item of $c(s_k)$. A query on these streams, $\mathcal{T}$, is a rooted AND-OR tree with L leaves $\mathcal{L} = \{l_1, \ldots, l_L\}$. Leaf l_j has success, resp. failure, probability p_j, resp. $q_j = 1 - p_j$, and requires the last $d_{l_j}^{s_k}$ items from each stream $s_k \in \mathcal{S}$. $d_{l_j}^{s_k}$ is zero if l_j does not require items from s_k. The objective is to compute the truth value of the root of the query tree by evaluating the leaves of the tree. Because each non-leaf node is either an OR or an AND operator, it may not be necessary to evaluate all the leaves due to *shortcircuiting*. In other words, as soon as any child node of an OR, resp. AND, operator evaluates to TRUE, resp. FALSE, the truth value of the operator is known and can be propagated toward the root. For a given query, we define a *schedule* as an evaluation order of the leaves of the query tree, represented as a sorted leaf sequence.

We define the *cost* of a schedule as the **expected value** of the sum of the costs incurred for all leaves that are evaluated before the root's truth value is determined. For instance, consider the query in Figure 7.1-(a), in which leaves are labeled l_1, l_2, l_3, and consider the schedule l_2, l_3, l_1. Query processing begins with the acquisition of the data items necessary for evaluating l_2, which has cost $4 \cdot c(B)$. With probability p_2, l_2 evaluates to TRUE, thus shortcircuiting the evaluation of l_3. Therefore, the expected evaluation cost of the OR operator is: $4 \cdot c(B) + q_2 \cdot c(C)$. If the OR operator evaluates to FALSE, which happens with probability $q_2 q_3$, then the evaluation of l_1 is shortcircuited. Otherwise, l_1 must be evaluated. The overall cost of the schedule is thus: $4 \cdot c(B) + q_2 \cdot c(C) + (1 - q_2 q_3) \cdot 5 \cdot c(A)$. Recall that this query tree corresponds to a *read-once* query.

The PAOTR problem consists in determining a schedule with minimum cost. For *read-once* queries the complexity of PAOTR is unknown for general AND-OR query trees, while optimal polynomial-time algorithms are known for AND trees [104] and DNF trees [105]. In this work, we focus on these two types of trees as well but for *shared* queries.

7.3 Evaluation of a schedule

Our overall objective is to study the problem of computing an optimal schedule for AND and DNF trees for *shared* queries. First, in this section we explain how the expected cost of a given schedule can be computed. This computation is non-trivial, as seen on an example (Section 7.3.1), but can be performed in polynomial time (Section 7.3.2).

7.3.1 Schedule evaluation examples

The *single-stream* DNF tree example

In this section, we illustrate on an example what is involved when computing the expected cost of a schedule. Consider the DNF tree in Figure 7.2 with three AND nodes, for four streams A, B, C, and D. For each leaf we indicate how many data items it requires from each stream and its probability of success. In this example each leaf requires a single data item from a stream. Since each leaf requires data items from a single stream this tree Leaves are labeled l_1 to l_7, in the order in which they appear in a given schedule. Computing the cost of a schedule is much more complicated than for *read-once* queries due to inter-leaf dependencies. Let $\mathcal{C}_j$ be the cost of evaluating leaf l_j, and $\mathcal{C}$ the overall cost of the schedule. We consider the 7 leaves one by one, in order:

Leaf l_1 – The first leaf is evaluated: $\mathcal{C}_1 = c(A)$.

Leaf l_2 – This is the first leaf in its AND, no AND has been fully evaluated so far, and l_2 is the first encountered leaf that requires stream B. Therefore, l_2 is always evaluated, requiring a data item from stream B: $\mathcal{C}_2 = c(B)$.

Leaf l_3 – This is the second leaf from its AND, no AND has been fully evaluated so far, and l_3 is the first encountered leaf that requires stream C. Therefore, a data item from C is acquired if and only if l_1 evaluates to TRUE: $\mathcal{C}_3 = p_1 c(C)$.

Leaf l_4 – This is the third leaf from its AND, no AND has been fully evaluated so far, and l_4 is the first encountered leaf that requires stream D. Therefore, one data item is acquired from D if and only if l_1 and l_3 both evaluate to TRUE: $\mathcal{C}_4 = p_1 p_3 c(D)$.

Leaf l_5 – This is the second leaf from its AND, and AND$_1$ has been fully evaluated so far. However, one of the leaves of that AND, l_3, requires a data item that is also needed by l_5, from stream C. If l_3 has been evaluated, then the evaluation cost of l_5 is 0 because the necessary data item from C has already been acquired and is available “for free” when evaluating l_5. If l_3 has

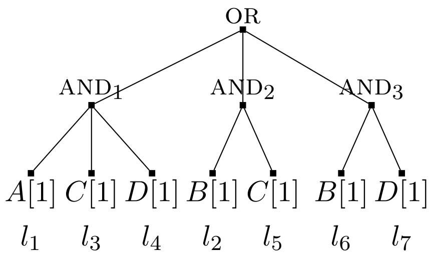


Figure 7.2: Example *single-stream* DNF tree.

not been evaluated (with probability $1 - p_1$), it means that AND$_1$ has evaluated to FALSE. Then, if l_2 has evaluated to TRUE, l_5 must be evaluated thus requiring the data item from stream C. We obtain $\mathcal{C}_5 = (1 - p_1)p_2c(C)$.
Leaf l_6 – Since l_2 is always evaluated the data item from stream B required by l_6 is always available for free: $\mathcal{C}_6 = 0$.
Leaf l_7 – This is the second leaf from its AND, and AND$_1$ and AND$_2$ have been fully evaluated so far. However, one of the leaves of AND$_1$, l_4, but none of those of AND$_2$, requires the data item that is needed by l_7 from stream D. Therefore, l_7 must be evaluated and its evaluation is not free if and only if l_4 has not been evaluated, AND$_2$ has evaluated to FALSE, and the evaluation of AND$_3$ went as far as l_7. Therefore, $\mathcal{C}_7 = (1 - p_1p_3)(1 - p_2p_5)p_6c(D)$.

Overall, we obtain the cost of the schedule:

$$\mathcal{TC} \quad = \quad c(A) + c(B) + (p_1 + (1 - p_1)p_2)c(C) + (p_1p_3 + (1 - p_1p_3)(1 - p_2p_5)p_6)c(D).$$

Given the complexity of the above cost computation on a small example, one might expect the PAOTR problem to be NP-complete for *shared* queries (recall that it is polynomial for *read-once* queries). We confirm this expectation in Section 7.5.

The *multi-stream* DNF tree example

Figure 7.3 shows a *multi-stream* DNF tree with three AND nodes, for five streams A, B, C, D, and E. Each leaf requires one or two data items from multiple streams. Leaves are labeled l_1 to l_6, in the order in which they appear in a given schedule. This example is meant to illustrate the difficulty of the PAOTR problem in the case of *multi-stream* DNF trees in the *shared* case. In particular, computing the cost of a schedule is much more complicated than in the *read-once single-stream* case due to inter-leaf dependencies and to the multiple access to several streams. Let $\mathcal{C}_j$ be the cost of evaluating leaf l_j, and $\mathcal{C}$ the overall cost of the schedule. We consider the 6 leaves one by one, in order:
Leaf l_1 – The first leaf is evaluated: $\mathcal{C}_1 = c(A) + c(C)$.
Leaf l_2 – This is the first leaf in its AND, no AND has been fully evaluated so far and since l_1 is always evaluated the first data item from stream C required by l_2 is always available for free. l_2 is the first encountered leaf that requires stream D. Therefore, l_2 is always evaluated, requiring a data item from stream D: $\mathcal{C}_2 = c(D)$.
Leaf l_3 – This is the second leaf in its AND, no AND has been fully evaluated so far, and l_3 is the first encountered leaf that requires stream B and stream E. Therefore, a data item from B and a data item from E are acquired if and only if l_1 evaluates to TRUE: $\mathcal{C}_3 = p_1(c(B)+c(E))$.
Leaf l_4 – This is the second leaf in its AND, and it requires a data item from stream D. This data item has already been acquired by l_2 and is available "for free" because l_2 is always

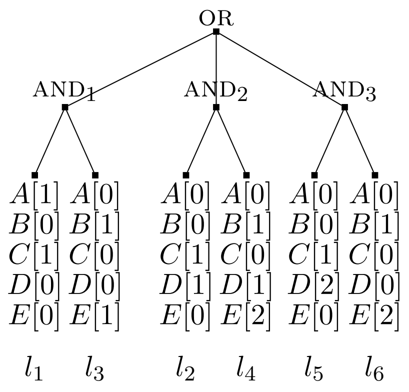


Figure 7.3: Example *multi-stream* DNF tree.

evaluated. l_4 also requires a data item from stream B and a data item from stream E. These data items are also required by leaf l_3 in AND$_1$, which has been fully evaluated. If l_3 has not been evaluated (with probability $1-p_1$), it means that AND$_1$ has evaluated to FALSE. Then, if l_2 has evaluated to TRUE (with probability p_2), l_4 must be evaluated thus requiring the data item from stream B and the first data item from stream E. Finally, l_3 is the first encountered leaf that requires the second data item from stream E, so if l_2 has evaluated to TRUE, then we must also require the second data item from stream E. We obtain $\mathcal{C}_4 = p_2[(1-p_1)(c(B)+c(E))+c(E)]$.
Leaf l_5 – Since l_2 is always evaluated, the data item from stream C and the first data item from stream D required by l_5 are always available for free. l_5 is the first encountered leaf that requires the second data item from stream D, so if AND$_1$ and AND$_2$ have evaluated to FALSE, then the second data item from D is acquired by l_5. We obtain $\mathcal{C}_5 = (1-p_1p_3)(1-p_2p_4)c(D)$
Leaf l_6 – This is the second leaf in its AND, and AND$_1$ and AND$_2$ have been fully evaluated so far. However, one of the leaves of AND$_1$, l_3, and one of the leaves of AND$_2$, l_4, require a data item from stream B and a data item from stream E that are needed by l_6. Leaf l_6 must be evaluated if l_5 has evaluated to TRUE (with probability p_5) and must acquire a data item from stream B and the first data item from stream E if and only if l_3 and l_4 have not been evaluated (with probability $(1-p_1)(1-p_2)$). l_6 must also acquire the second data item from stream E if and only if l_4 has not been evaluated (with probability $(1-p_2)$). We obtain $\mathcal{C}_6 = p_5[(1-p_1)(1-p_2)(c(B)+c(E))+(1-p_2)c(E)]$.

Overall, we obtain the cost of the schedule:

$$\begin{aligned}\mathcal{TC} &= c(A)+c(C)+[p_1+(1-p_1)(p_2+p_5(1-p_2))]c(B)+\\ &+ (1+(1-p_1p_3)(1-p_2p_4))c(D)+[p_1+(2-p_1)(p_2+p_5(1-p_2))]c(E)\end{aligned}$$

Given the complexity of the above cost computation, one might expect the PAOTR problem to be NP-complete in the *shared multi-stream* case. We confirm this expectation for AND trees in Section 7.4.3.

7.3.2 Schedule evaluation algorithm

Consider a DNF tree with N AND nodes, indexed $i = 1,\dots,N$. As defined in Section 7.2 the set of leaves is denoted by $\mathcal{L}$ and has cardinal L. To capture the structure of the DNF tree we modify the leaf notation in Section 7.2 as follows. AND node i has m_i leaves, denoted by $l_{i,j}$, $j = 1,\dots,m_i$. The probability of success of leaf $l_{i,j}$ is denoted by $p_{i,j}$. The query is over S streams, s_s, $s = 1, \ldots, S$ and each leaf can require data from multiple streams as in the more general *multi-stream* case. The cost per data item of s_s is denoted by $c(s_s)$. We define the "t-th data item" of a stream as the data item produced t time-steps ago, so that the first data item is the one produced most recently, the second is the one produced before the first, etc. In this manner, when we say that leaf $l_{i,j}$ requires $d^{s_s}_{l_{i,j}}$ data items from stream s_s it means that it requires all t-th data items of the stream s_s for $t = 1, 2, \ldots, d^{s_s}_{l_{i,j}}$. Finally, we consider a schedule ξ, which is an ordering of the leaves, and use $l_{r,t} \prec l_{u,v}$ to indicate that leaf $l_{r,t}$ occurs before leaf $l_{u,v}$ in ξ.

Given the above, we define $\mathcal{L}_{s,t}$ as the set of the leaves that require the t-th data item from stream s_s and that are the first of their respective AND nodes to require that data item. Formally, we have:

$$\mathcal{L}_{s,t} = \left\{ l_{i,j} \in \mathcal{L} \;\middle|\; d^{s_s}_{l_{i,j}} \geq t, \text{ and } (\forall r \neq j, d^{s_s}_{l_{i,r}} < t \text{ or } l_{i,j} \prec l_{i,r}) \right\}.$$

We also define $\mathcal{A}_{i,j}$, the index set of all AND nodes that have been fully evaluated before leaf $l_{i,j}$ is evaluated, as:

$$\mathcal{A}_{i,j} = \{k \mid m_k = |\{l_{k,r} | l_{k,r} \prec l_{i,j}\}|\}.$$

If we use $\mathcal{C}_{l_{i,j},s,t}$ to denote the expected cost of retrieving the t-th data item of stream s_s when evaluating leaf $l_{i,j}$, then the total cost $\mathcal{C}$ of the schedule ξ is:

$$\mathcal{C} = \sum_{l_{i,j} \in \xi} \left(\sum_{s=1}^{S} \sum_{t=1}^{d^{s_s}_{l_{i,j}}} \mathcal{C}_{l_{i,j},s,t} \right). \tag{7.1}$$

The following proposition gives $\mathcal{C}_{l_{i,j},s,t}$.

Proposition 14. *Given a leaf $l_{i,j}$ that does not require the t-th data item from stream s_s, then $\mathcal{C}_{l_{i,j},s,t} = 0$. Otherwise, if there exists r such that $l_{i,r} \prec l_{i,j}$ and $l_{i,r} \in \mathcal{L}_{s,t}$, then $\mathcal{C}_{l_{i,j},s,t} = 0$, else:*

$$\begin{aligned}
\mathcal{C}_{l_{i,j},s,t} = \quad & \prod_{\substack{l_{r,v} \in \mathcal{L}_{s,t} \\ l_{r,v} \prec l_{i,j}}} \left(1 - \prod_{l_{r,u} \prec l_{r,v}} p_{r,u} \right) \\
\times \quad & \prod_{\substack{a \in \mathcal{A}_{i,j} \\ \nexists r,\ l_{a,r} \in \mathcal{L}_{s,t}}} \left(1 - \prod_{r=1}^{m_a} p_{a,r} \right) \\
\times \quad & \left(\prod_{l_{i,u} \prec l_{i,j}} p_{i,u} \right) \times c(s_s).
\end{aligned}$$

Proof. Consider a schedule ξ, a stream s_s, and an integer t. Consider a leaf in that schedule, $l_{i,j}$, which requires the t-th data item from stream s_s. Let us prove the first part of the proposition. If leaf $l_{i,j}$ does not require the t-th data item from stream s_s, then the acquisition cost is 0. Otherwise, if a leaf $l_{i,r}$ (i.e., a leaf in the same AND node as $l_{i,j}$) occurs before $l_{i,j}$ in ξ ($l_{i,r} \prec l_{i,j}$) and requires the t-th item from stream s_s (i.e., $l_{i,r} \in \mathcal{L}_{s,t}$), then there are two possibilities. Either $l_{i,r}$ has been evaluated, in which case the evaluation of $l_{i,j}$ uses a data item that has already been acquired previously, hence a cost of 0. Or $l_{i,k}$ has not been evaluated, meaning that its evaluation was shortcircuited. In this case the AND node has evaluated to FALSE and the evaluation of $l_{i,j}$ is also shortcircuited and the cost is 0.

The second part of the proposition shows the expected cost as a product of three factors, each of which is a probability, and a fourth factor, $c(s_s)$, which is the cost of acquiring the data item from the stream. The interpretation of the expression for $\mathcal{C}_{l_{i,j},s,t}$ is as follows: a leaf must acquire the t-th item from stream s_s if and only if (i) the item has not been previously acquired; and (ii) no AND node has already evaluated to TRUE; and (iii) no leaf in the same AND node has already evaluated to FALSE. We explain the computation of these three probabilities hereafter.

The first factor is the probability that none of the leaves that precede $l_{i,j}$ in ξ and that require the t-th item from stream s_s have been evaluated. Such a leaf $l_{r,s}$ is evaluated if all the leaves in the same AND node that precede it in the schedule have evaluated to TRUE, which happens with probability $\prod_{l_{r,u} \prec l_{r,s}} p_{r,u}$. Hence, the expression for the first factor.

The second factor is the probability that none of the AND nodes that have been fully evaluated so far has evaluated to TRUE, since if this were the case the evaluation of $l_{i,j}$ would not be needed, leading to a cost of 0. Given an AND node in $\mathcal{A}_{i,j}$, say the k-th AND node, the probability that it has evaluated to TRUE is $\prod_{r=1}^{m_k} p_{k,r}$. This is true except if one of the leaves of that AND node belongs to $\mathcal{L}_{s,t}$. The first factor assumes that that leaf was not evaluated and, therefore, that that entire AND node was not evaluated. Hence, the expression for the second factor.

The third factor is the probability that all the leaves in the same AND node as $l_{i,j}$ that have been evaluated have evaluated to TRUE. Because we are in the second case of the proposition, none of these leaves requires the t-th item of stream s_s. All these leaves must evaluate to TRUE, otherwise the evaluation of $l_{i,j}$ would be shortcircuited, for a cost of 0. Hence, the expression for the third factor. ■

The expected cost of a schedule ξ can be computed from Eq. (7.1) and Proposition 14. We now derive the complexity of carrying out this computation. Recall that L is the total number of leaves in the tree. Let D be the maximum number of required data items over all streams. More formally, $L = \sum_{i=1}^{N} m_i$ and $D = \max_{1 \le i \le N, 1 \le j \le m_i} d_{l_{i,j}}$. To compute all the sets $\mathcal{L}_{s,t}$ we need to scan the leaves of each AND node according to schedule ξ while recording the maximum number of items required from each stream. This can be done with complexity $O(L)$. Each set $\mathcal{L}_{s,t}$ contains at most N leaves. Computing all the sets $\mathcal{A}_{i,j}$ is also done through a traversal of the set of leaves, for an overall cost of $O(L+N^2)$ (because the sets $\mathcal{A}_{i,j}$ take at most N distinct values and each contains at most N elements). Computing all the product of probabilities used in the computation of all the $\mathcal{C}_{l_{i,j},s,t}$ can also be done in a single traversal of the set of leaves. Once all these precomputations are done, the first term in the expression of $\mathcal{C}_{l_{i,j},s,t}$ can be computed in $O(N)$, the second in $O(N^2)$, and the third one in $O(1)$. We compute $\mathcal{C}_{l_{i,j},s,t}$ for all the streams required by the leaf $l_{i,j}$ and the maximum number of streams required by a leaf is S. Overall the cost of a schedule can be evaluated with complexity

$$O(LSDN^2).$$

7.4 AND trees

In this section we focus on AND tree for *shared* queries. We first show that the optimal algorithm for *read-once* queries is no longer optimal for *shared* queries (Section 7.4.1). We develop an optimal greedy algorithm in the *single-stream* case (Section 7.4.2). We show that the problem is NP-complete in the *multi-stream* case, for which we propose a heuristic that we show to be close to the optimal for small instances (Section 7.4.3).

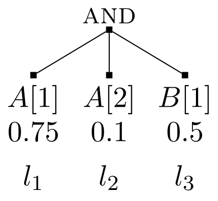


Figure 7.4: Example *shared* AND tree for which the *read-once* algorithm in [104] is not optimal.

7.4.1 Is the optimal *read-once* algorithm still optimal?

One valid question is whether the algorithm in [104], which is optimal for *read-once* queries, is still optimal for *shared* queries. It turns out that it is not, and in this section we provide a counter-example. Consider the AND tree depicted in Figure 7.4 with three leaves labeled l_1, l_2, and l_3, for two streams A and B. For each leaf (l_i), we indicate the stream, the number of data items needed from that stream to evaluate the leaf, and the success probability (p_i). For instance, leaf l_2 requires $d^A_{l_2} = 2$ items from stream A and evaluates to TRUE with probability $p_2 = 0.1$. We assume that retrieving a data item from any stream has unitary cost. There are 6 possible schedules for this tree, each schedule corresponding to one of the 3! orderings of the leaves. The algorithm in [104] sorts the leaves by non-decreasing $d^s_{l_j} c(s)/q_j$, where s is the only stream from which l_j requires data items. Because $\frac{1\times c(A)}{q_1} = \frac{1}{1-0.75} = 4$, $\frac{2\times c(A)}{q_2} = \frac{2}{1-0.1} \approx 2.22$, and $\frac{1\times c(B)}{q_3} = \frac{1}{1-0.5} = 2$, this algorithm schedules leaf l_3 first. There are two possible schedules with l_3 as the first leaf:

— l_3, l_1, l_2 whose cost is: $c(B) + p_3 \times (c(A) + p_1 \times c(A)) = 1 + 0.5 \times (1 + 0.75 \times 1) = 1.875$; and

— l_3, l_2, l_1 whose cost is: $c(B) + p_3 \times (2 \times c(A) + p_2 \times 0 \times c(A)) = 1 + 0.5 \times (2 + 0.1 \times 0) = 2$.

However, another schedule, l_1, l_2, l_3, has a lower cost: $c(A) + p_1 \times (c(A) + p_2 \times c(B)) = 1 + 0.75 \times (1 + 0.1 \times 1) = 1.825$. Therefore, the optimal algorithm for the PAOTR problem for *read-once* AND trees is no longer optimal for *shared* AND trees.

7.4.2 The *single-stream* case

In this section we give an optimal algorithm for solving the problem for *shared* AND trees in the *single-stream* case. Like the algorithm in [104] for *read-once* queries, our algorithm is greedy. But it compares the ratios of cost to failure probability of all sequences of leaves that use the same stream, instead of only considering pair-wise leaf comparisons. We begin with a preliminary result on the optimal ordering of leaves that use the same stream.

Ordering same-stream leaves

In the example given in Section 7.4.1 we consider two schedules that begin with leaf l_3. In the first schedule leaf l_1 precedes l_2, while the converse is true in the second schedule. Leaf l_1 requires one data item from stream A, while leaf l_2 requires two data items from the same stream. Therefore the first schedule is always preferable to the second schedule: if we evaluate l_1 before l_2 and if l_1 evaluates to FALSE, then there is no need to retrieve the second data item and the cost is lowered. A general result can be obtained:

Proposition 15. *Consider an* AND *tree and a leaf l_i that requires $d^s_{l_i} > 0$ data items from a stream s. In an optimal schedule, l_i is scheduled before any leaf l_j that requires $d^s_{l_j} > d^s_{l_i}$ data items from stream s.*

Proof. We prove the proposition by contradiction. Consider an AND-tree and two leaves in the tree l_1 and l_2 that require data items from the same stream s such that $d^s_1 > d^s_2$. We terms these leaves "inverted" because the earlier one, l_1, requires more data items than the later one, l_2. Assume that there is an optimal schedule ξ in which l_1 is scheduled before l_2. Without loss of generality, we assume that l_1 is the first leaf in the schedule that is part of an inverted pair of leaves (if not, consider the earliest such leaf). Evaluating l_2 has always cost zero in this schedule because all data items required by l_2 are also required by l_1.

The sequence of leaves in ξ can be written as: $l_{b_1}, \ldots, l_{b_t}, l_1, l_{m_1}, \ldots, l_{m_u}, l_2, l_{a_1}, \ldots, l_{a_v}$. The cost $\mathcal{C}$ of ξ can be written:

$$\mathcal{C} = X + \mathcal{P}_b \cdot (d^s_1 - d^s_{LB})c(s) + \mathcal{P}_b \cdot p_1 \cdot Y + \mathcal{P}_b \cdot p_1 \cdot \mathcal{P}_m \cdot 0 + \mathcal{P}_b \cdot p_1 \cdot \mathcal{P}_m \cdot p_2 \cdot Z$$

where

— $\mathcal{P}_b = \prod_{i=1}^{t} p_{b_i}$ and $\mathcal{P}_m = \prod_{i=1}^{u} p_{m_i}$;

— X is the expected cost of evaluating leaves $l_{b_1}, \ldots, l_{b_t}$ in that order;

— Y is the expected cost of evaluating leaves $l_{m_1}, \ldots, l_{m_u}$ in that order if leaves $l_{b_1}, \ldots, l_{l_t}$ and l_1 all evaluate to TRUE;

— Z is the expected cost of evaluating leaves $l_{a_1}, \ldots, l_{a_v}$ in that order if leaves $l_{b_1}, \ldots, l_{b_t}$, l_1, $l_{l_1}, \ldots, l_{m_u}$, and l_2 all evaluated to TRUE;

— $d^s_{LB} = \max_{i=1,\ldots,t}(d^s_{b_i})$, or the number of elements of stream s that have been acquired after evaluating leaves $l_{b_1}, \ldots, l_{b_t}$.

Because l_1 and l_2 are the first two inverted leaves in ξ, $d^s_1 - d^s_{LB}$ is non-negative (otherwise a leaf among $l_{b_1}, \ldots, l_{b_t}$ and leaf l_1 would be inverted).

We now construct another schedule, ξ', as $l_{b_1}, \ldots, l_{b_t}, l_2, l_1, l_{m_1}, \ldots, l_{m_u}, l_{a_1}, \ldots, l_{a_v}$. The expected cost $\mathcal{C}$' of ξ' can then be written as:

$$\mathcal{C}' = X + \mathcal{P}_b \cdot (d^s_2 - d^s_{LB})c(s) + \mathcal{P}_b \cdot p_2(d^s_1 - d^s_2)c(s) + \mathcal{P}_b \cdot p_2 \cdot p_1 \cdot Y + \mathcal{P}_b \cdot p_2 \cdot p_1 \cdot \mathcal{P}_m \cdot Z$$

Because l_1 and l_2 are the first two inverted leaves in ξ, $d^s_2 - d^s_{LB}$ is non-negative (otherwise a leaf among $l_{b_1}, \ldots, l_{b_t}$ and leaf l_2 would be inverted). Computing the difference of the costs of both schedules yields:

$$\mathcal{C} - \mathcal{C}' = \mathcal{P}_b(1 - p_2)\left((d^s_1 - d^s_2)c(s) + p_1 Y\right)$$

$\mathcal{C} - \mathcal{C}'$ is strictly positive because all costs are positives, all probabilities are between 0 and 1, and because $d^s_1 > d^s_2$ by assumption. This contradicts the optimality of ξ. ■

Optimal schedule

Consider an AND tree with L leaves, $l_1, \ldots, l_L$, for S streams, $s_1, \ldots, s_S$. We define $\mathcal{L}_k = \{l_j \mid d^{s_k}_{l_j} > 0\}$, i.e., the set of leaves that require data items from stream s_k. We propose a greedy algorithm, SINGLESTREAMGREEDY (Algorithm 6). This algorithm, which is implemented recursively for clarity of presentation, takes as input the $\mathcal{L}_k$ sets, an initially empty schedule ξ, and an array of S integers, *NumItems*, whose elements are all initially set to zero. This array is used to keep track, for each stream, of how many data items from that stream have been retrieved in the schedule so far. Each call to the algorithm appends to the schedule a sequence of leaves that require data items from the same stream, in increasing order of number of data items required. The algorithm stops when all leaves have been scheduled. The algorithm first loops through all the streams (the k loop). For each stream, the algorithm then loops over all the leaves that use that stream, taken in increasing order of the number of items required. For each such leaf the algorithm computes the ratio (variable *Ratio*) of cost to probability of failure of the sequence of leaves up to that leaf. The leaf with the minimum such ratio is selected (leaf l_{j_0} in the algorithm, which requires $d_{l_{j_0}}^{s_{k_0}}$ data items from stream s_{k_0}). In the last loop of the algorithm, all unscheduled leaves that require $d_{l_{j_0}}^{s_{k_0}}$ or fewer data items from stream s_{k_0} are appended to the schedule in increasing order of the number of required data items.

```
Algorithm 6: SingleStreamGreedy($\{\mathcal{L}_1, ..., \mathcal{L}_S\}, \xi, NumItems$)
1  if $\cup_{i=1}^{S} \mathcal{L}_i = \varnothing$ then return $\xi$ $MinRatio \leftarrow +\infty$;
2  for $k = 1$ to $S$ do
3      $Cost \leftarrow 0$;
4      $Proba \leftarrow 1$;
5      $Num \leftarrow NumItems[k]$;
6      for $l_j \in \mathcal{L}_k$ by non-decreasing $d_{l_j}^{s_k}$ do
7          $Cost \leftarrow Cost + Proba \times (d_{l_j}^{s_k} - Num) \times c(k)$;
8          $Proba \leftarrow Proba \times p_j$;
9          $Num \leftarrow d_{l_j}^{s_k}$;
10         $Ratio \leftarrow \frac{Cost}{(1-Proba)}$;
11         if $Ratio < MinRatio$ then
12             $MinRatio \leftarrow Ratio$;
13             $j_0 \leftarrow j$; $k_0 \leftarrow k$;
14 for $l_j$ in $\mathcal{L}_{k_0}$ by non-decreasing $d_{l_j}^{s_{k_0}}$ do
15     if $d_{l_j}^{s_{k_0}} \leq d_{l_{j_0}}^{s_{k_0}}$ then
16         $\xi \leftarrow \xi \cdot l_j$;
17         $\mathcal{L}_{k_0} \leftarrow \mathcal{L}_{k_0} \setminus \{l_j\}$;
18 $NumItems[k_0] \leftarrow d_{l_{j_0}}^{s_{k_0}}$;
19 return SingleStreamGreedy($\{\mathcal{L}_1, ..., \mathcal{L}_S\}, \xi, NumItems$)
```

Theorem 9. SingleStreamGreedy *is optimal for the* shared *PAOTR problem for* AND *trees.*

Proof sketch. We prove the theorem by contradiction. We assume that there exists an instance for which the schedule produced by SingleStreamGreedy, ξ_{greedy}, is not optimal. Among the optimal schedules, we pick a schedule, ξ_{opt}, which has the longest prefix $\mathbb{P}$ in common with ξ_{greedy}. We consider the first decision taken by SingleStreamGreedy that schedules a leaf that does not belong to $\mathbb{P}$. Let us denote by $l_{\sigma(1)}$, ..., $l_{\sigma(k)}$ the sequence of leaves scheduled by this decision. The first leaves in this sequence may belong to $\mathbb{P}$. Let $\mathbb{P}'$ be $\mathbb{P}$ minus the leaves $l_{\sigma(1)}$, ..., $l_{\sigma(k)}$. Then, ξ_{greedy} can be written as:

$$\xi_{greedy} = \mathbb{P}', l_{\sigma(1)}, ..., l_{\sigma(k)}, \mathbb{S}.$$

In turn, ξ_{opt} can be written $\xi_{opt} = \mathbb{P}', \mathbb{Q}, \mathbb{R}$ where $l_{\sigma(k)}$ is the last leaf of $\mathbb{Q}$. In other words, $\mathbb{Q}$ can be written $L_1, l_{\sigma(1)}, ..., L_k, l_{\sigma(k)}$, where each sequence of leaves L_i, $1 \leq i \leq k$, may be empty. We can write:

$$\xi_{opt} = \mathbb{P}', L_1, l_{\sigma(1)}, ..., L_k, l_{\sigma(k)}, \mathbb{R}.$$

From ξ_{greedy} and ξ_{opt}, we build a new schedule, ξ_{new}, defined as

$$\xi_{new} = \mathbb{P}', l_{\sigma(1)}, ..., l_{\sigma(k)}, L_1, ..., L_k, \mathbb{R}.$$

$\mathbb{P}'$, $l_{\sigma(1)}$, ..., $l_{\sigma(k)}$ is a prefix to both ξ_{greedy} and ξ_{new}. This prefix is strictly larger than $\mathbb{P}$ since $\mathbb{P}$ does not contain $l_{\sigma(k)}$. We compute the cost of ξ_{new} and show that it is no larger than that of ξ_{opt}, thus showing that ξ_{new} is optimal and has a longer prefix in common with ξ_{greedy} than ξ_{new}, which is a contradiction. This computation is lengthy and technical and the full proof is provided in the Appendix 7.7.

■

The complexity of SingleStreamGreedy is $O(L^2)$. Indeed, the sets $\mathcal{L}_1$, ..., $\mathcal{L}_S$ are built and sorted in $O(L\log(L))$ time and there are at most L recursive calls to SingleStreamGreedy, each having a cost proportional to the number of leaves remaining in the AND tree.

One may wonder how the optimal algorithm for *read-once* queries [104], which simply sorts the leaves by increasing $d^s_{l_j} c(s)/q_j$, fares for *shared* queries. In other terms, is SingleStreamGreedy really needed in practice? Figure 7.5 shows results for a set of randomly generated AND trees. We define the *sharing ratio*, ρ, of a tree as the expected number of leaves that use the same stream, i.e., the total number of leaves divided by the number of streams. For a given number of leaves $L = 2, \ldots, 20$ and a given sharing ratio $\rho = 1, 5/4, 4/3, 3/2, 2, 3, 4, 5, 10$, we generate 1,000 random trees for a total of 157,000 random trees (note that ρ cannot be larger than the number of leaves). Leaf success probabilities, numbers of data items needed at each leaf, and per data item costs are sampled from uniform distributions over the intervals $[0, 1]$, $[1, 5]$, and $[1, 10]$, respectively. For each tree we compute the cost achieved by the algorithm in [104] and that achieved by our optimal algorithm. Figure 7.5 plots these costs for all instances, sorted by increasing optimal cost. Due to this sorting, the large number of samples, and the limited resolution, the set of points for the optimal algorithm appears as a curve while the set of points for the algorithm in [104] appears as a cloud of points. These results show that the algorithm in [104] can lead to costs up to 1.86 times larger than the optimal. It leads to costs more than 10% larger for 19.54% of the instances, and more than 1% larger for 60.20% of the instances. The two algorithms lead to the same cost for only 11.29% of the instances. We conclude that, for *shared* queries, SingleStreamGreedy provides substantial improvements over the optimal algorithm for *read-once* queries.

7.4.3 The *multi-stream* case

In this section, we first show the NP-completeness of determining the optimal schedule for a *multi-stream* AND tree. Next we show how to extend the greedy algorithm of the *single-stream* case. While no longer optimal, this extended greedy algorithm is close to the optimal in practice and thus proves useful for designing heuristics.

The *multi-stream* case is NP-complete

Definition 1 (AND-Multi-Decision)**.** Given a *multi-stream* AND tree and a cost bound K, is there a schedule whose expected cost does not exceed K?

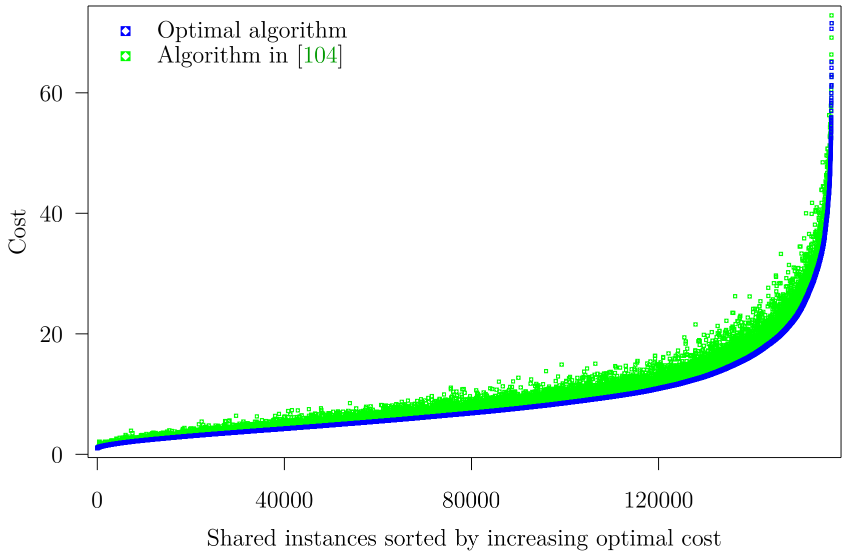


Figure 7.5: Cost achieved by the algorithm in [104] and that achieved by the optimal SINGLESTREAMGREEDY algorithm, for 157,000 random AND tree instances sorted by increasing optimal cost.

Theorem 10. AND-MULTI-DECISION *is NP-complete.*

Proof. The problem is clearly in NP: given a schedule, i.e., an ordering of the leaves, one can compute its expected cost in polynomial time (using the method given in Section 7.3) and compare it to K. The NP-completeness is obtained by reduction from 2-PARTITION [94]. Let $\mathcal{I}_1$ be an instance from 2-PARTITION: given a set $\{a_1, ..., a_n\}$ and $S = \sum_{i=1}^{n} a_i$, does there exist a subset I such that $\sum_{i \in I} a_i = \frac{S}{2}$? We assume that S is even, otherwise there is no solution. The size of $\mathcal{I}_1$ is $O(n \times \log M)$, where $M = \max_{1 \le i \le n}\{a_i\}$. Without loss of generality, we assume that $M \geq 10$. We construct the following instance $\mathcal{I}_2$ of AND-MULTI-DECISION:

— We consider an AND tree with $n+1$ leaves ℓ_i, $1 \le i \le n+1$. The set of streams is $\mathcal{S} = \{A_1, \dots, A_n, B\}$. The cost of stream $s_i = A_i$ for $i \le n$ is $c(i) = \frac{1}{2Z}$, where Z is some large constant defined hereafter. The cost of stream $s_{n+1} = B$ is $c(n+1) = C_0$, where $C_0 \approx \frac{1}{2}$ is a constant defined hereafter.

— The first n leaves have a single stream: for $1 \le i \le n$, leaf ℓ_i accesses $2a_i$ elements of stream A_i, so that the cost of evaluating leaf ℓ_i (without re-use) is $\frac{a_i}{Z}$. The success probability of leaf ℓ_i is

$$p_i = 1 - \frac{a_i}{Z} - \beta \frac{a_i^2}{Z^2},$$

where $\beta \approx \frac{1}{2}$ is a constant defined hereafter.

— Leaf ℓ_{n+1} accesses all $n+1$ streams: one element of the stream B, and a_i elements of each of the n streams A_i. The cost of evaluating leaf ℓ_{n+1} (assuming no re-use of data items acquired during the evaluation of other leaves) is $C = C_0 + \sum_{i=1}^{n} \frac{a_i}{2Z} = C_0 + \frac{S}{2Z}$. The success probability of leaf ℓ_{n+1} is $p_{n+1} = \varepsilon$. Constant ε is chosen to be very small, see below. Intuitively, C would be the cost of a schedule evaluating leaf ℓ_{n+1} first, and thereby terminating the evaluation, when ε becomes negligible.

— The bound on the expected evaluation cost is $K = C\left(1 - \frac{S^2}{8Z^2}\right) + \frac{1}{9Z^2}$.

To finalize the description of $\mathcal{I}_2$, we define the constants as follows:

— $Z = 10\left((n+1)3^n + n^3\right) M^3$,

— $C_0 = \frac{Z}{2Z-S} - \frac{S}{2Z}$, so that $C = \frac{Z}{2Z-S}$,

— $\beta = \frac{1-C}{2C}$,

— $\varepsilon = \frac{1}{(n+1)^2 90 Z^2}$.

The size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$: the greatest value in $\mathcal{I}_2$ is Z and $\log(Z)$ is linear in $(n + \log M)$. Because Z is very large relatively to $S \leq nM$, we do have that C, C_0, and β are all close to $\frac{1}{2}$. We only use that these constants are all nonnegative, and that $\beta \leq 1$ and $C \leq 1$, in the following derivation, where we bound the expected cost of an arbitrary evaluation of the AND tree. Then, using this derivation, we prove that $\mathcal{I}_1$ has a solution I if and only if $\mathcal{I}_2$ does.

Let us start with the cost of an arbitrary evaluation of the AND tree. In such an evaluation, we evaluate some (possibly none) of the first n leaves before evaluating leaf ℓ_{n+1}. Then, because ε is small, we can compute an approximation of the cost as follows: we assume that the schedule terminates after leaf ℓ_{n+1}, because its success probability is close to 0. We will bound the difference between this approximation and the actual cost later on.

Let $I = \{\ell_{\sigma(1)}, \ell_{\sigma(2)}, \ldots, \ell_{\sigma(k)}\}$ be the subset, of cardinal k, of leaves that are evaluated, in that order, before leaf ℓ_{n+1}. Let C be the approximated cost of the schedule (terminating after completion of leaf ℓ_{n+1}). To simplify notations, we let $x_i = a_{\sigma(i)}$ and $r_i = p_{\sigma(i)}$ for $1 \leq i \leq k$. By definition:

$$C = \sum_{i=1}^{k} \frac{x_i}{Z} \prod_{1 \leq j < i} r_j + \left(C - \sum_{i=1}^{k} \frac{x_i}{2Z}\right) \prod_{1 \leq j \leq k} r_j .$$

Note that the cost of leaf ℓ_{n+1} has been reduced from its original value, due to the sharing of the streams whose index is in I. To evaluate C, we start by approximating

$$\prod_{1 \leq j < i} r_j = \prod_{1 \leq j < i} \left(1 - \frac{x_j}{Z} - \beta \frac{x_j^2}{Z^2}\right).$$

Let

$$F_i = 1 - \sum_{j=1}^{i-1} \frac{x_j}{Z} - \beta \sum_{j=1}^{i-1} \frac{x_j^2}{Z^2} + \sum_{1 \leq j_1 < j_2 < i} \frac{x_{j_1} x_{j_2}}{Z^2}.$$

We have

$$\left| \left(\prod_{1 \leq j < i} r_j \right) - F_i \right| \leq \frac{3^n M^3}{Z^3}. \tag{7.2}$$

To see this, we have kept in F_i all terms of the product $\prod_{1 \leq j < i} r_j$ whose denominators include a factor strictly inferior to Z^3. The other terms of the product are bounded (in absolute value) by M^3/Z^3, because $\beta \leq 1$, $x_j \leq M$, and $M \leq Z$. There are at most $3^{i-1} \leq 3^n$ such terms. Hence the desired bound in Equation (7.2). Letting

$$G = \sum_{i=1}^{k} \frac{x_i}{Z} - \sum_{1 \leq j_1 < j_2 \leq k} \frac{x_{j_1} x_{j_2}}{Z^2},$$

we prove similarly that

$$\left| \left(\sum_{i=1}^{k} \frac{x_i}{Z} \prod_{1 \leq j < i} r_j \right) - G \right| \leq \frac{n 3^n M^3}{Z^3}. \tag{7.3}$$

Indeed, there are $k \leq n$ terms in the sum, each of them being bounded as before. We deduce from Equations (7.2) and (7.3), using $C \leq 1$, that

$$\left| C - \left(G + (C - \sum_{i=1}^{k} \frac{x_i}{2Z}) F_{k+1} \right) \right| \leq \frac{(n+1)3^n M^3}{Z^3}. \tag{7.4}$$

Now, we aim at simplifying $H = G + (C - \sum_{i=1}^{k} \frac{x_i}{2Z}) F_{k+1}$ by dropping terms whose denominator is Z^3. We have

$$H = \sum_{i=1}^{k} \frac{x_i}{Z} - \sum_{1 \leq j_1 < j_2 \leq k} \frac{x_{j_1} x_{j_2}}{Z^2} + \left(C - \sum_{i=1}^{k} \frac{x_i}{2Z} \right) \left(1 - \sum_{j=1}^{k} \frac{x_j}{Z} - \beta \sum_{j=1}^{k} \frac{x_j^2}{Z^2} + \sum_{1 \leq j_1 < j_2 \leq k} \frac{x_{j_1} x_{j_2}}{Z^2} \right).$$

Defining

$$\tilde{H} = C + \frac{1-2C}{2Z} \sum_{i=1}^{k} x_i + \frac{1}{2Z^2} \left(\sum_{i=1}^{k} x_i \right)^2 + \frac{C-1}{Z^2} \sum_{1 \leq j_1 < j_2 \leq k} x_{j_1} x_{j_2} - \frac{\beta C}{Z^2} \sum_{i=1}^{k} x_i^2,$$

we derive (using $\beta \leq 1$) that:

$$\left| H - \tilde{H} \right| \leq \left| \frac{1}{2Z^3} \left(\sum_{i=1}^{k} x_i \right) \left(\sum_{1 \leq j_1 < j_2 \leq k} x_{j_1} x_{j_2} + \sum_{i=1}^{k} x_i^2 \right) \right|.$$

Hence,

$$\left| H - \tilde{H} \right| \leq \frac{n^3 M^3}{Z^3}. \tag{7.5}$$

Developing $(\sum_{i=1}^{k} x_i)^2 = \sum_{i=1}^{k} x_i^2 + 2 \sum_{1 \leq j_1 < j_2 \leq k} x_{j_1} x_{j_2}$ in $\tilde{H}$, we obtain

$$\tilde{H} = C + \frac{1-2C}{2Z} \sum_{i=1}^{k} x_i + \frac{C}{Z^2} \sum_{1 \leq j_1 < j_2 \leq k} x_{j_1} x_{j_2} + \frac{1 - 2\beta C}{2Z^2} \sum_{i=1}^{k} x_i^2.$$

We have chosen the constants C and β so that $\tilde{H}$ can be reduced to

$$\tilde{H} = C + \frac{C}{2Z^2} \left(\left(\frac{S}{2} - \sum_{i=1}^{k} x_i \right)^2 - \frac{S^2}{4} \right). \tag{7.6}$$

Indeed, we have $\frac{1-2C}{2Z} = \frac{-SC}{2Z^2}$, and $C = 1 - 2C\beta$. Altogether, we derive from Equations (7.4) to (7.6) that

$$\left| C - C \left(1 - \frac{S^2}{8Z^2} \right) - \frac{C}{2Z^2} \left(\frac{S}{2} - \sum_{i=1}^{k} x_i \right)^2 \right| \leq \frac{\left((n+1)3^n + n^3 \right) M^3}{Z^3} = \frac{1}{10Z^2}. \tag{7.7}$$

Finally, we coarsely bound the difference between the actual cost *Cost* of the schedule and the approximated cost C. The actual probability of evaluating some other leaves after leaf ℓ_{n+1} is ε, there are at most n such leaves, whose individual cost does not exceed $\frac{M}{2Z}$. We get a difference bounded by $n \frac{M}{2Z} \varepsilon$, from which we derive

$$|Cost - C| \leq n \frac{M}{2Z} \varepsilon \leq (n+1)^2 \varepsilon = \frac{1}{90Z^2}. \tag{7.8}$$

We could easily tighten the bound in Equation (7.8), but we will keep the same notations to derive a similar bound in the proof of Theorem 12.

Combining Equations (7.7) and (7.8), we finally derive that

$$\left| Cost - C\left(1 - \frac{S^2}{8Z^2}\right) - \frac{C}{2Z^2}\left(\frac{S}{2} - \sum_{i=1}^{k} x_i\right)^2 \right| \leq \frac{1}{9Z^2}. \tag{7.9}$$

We now prove that $\mathcal{I}_1$ has a solution I if and only if $\mathcal{I}_2$ does. Suppose first that $\mathcal{I}_1$ has a solution I: $\sum_{i\in I} a_i = \frac{S}{2}$. We evaluate the leaves whose indices are in I before evaluating leaf ℓ_{n+1}, followed by the remaining leaves in any order. Let *Cost* be the cost of this evaluation. From Equation (7.9), we have

$$\left| Cost - C\left(1 - \frac{S^2}{8Z^2}\right) \right| \leq \frac{1}{9Z^2}.$$

Hence, $Cost \leq C\left(1 - \frac{S^2}{8Z^2}\right) + \frac{1}{9Z^2} = K$, thereby providing a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution whose cost is $Cost \leq K$, and let I denote the (index) set of leaves that are evaluated before leaf ℓ_{n+1}. If (by contradiction) we have $\sum_{i\in I} a_i \neq \frac{S}{2}$, then $\left(\frac{S}{2} - \sum_{i=1}^{k} x_i\right)^2 \geq 1$, and Equation (7.9) shows that

$$Cost \geq C\left(1 - \frac{S^2}{8Z^2}\right) + \frac{C}{2Z^2} - \frac{1}{9Z^2} = K + \frac{9C-4}{9Z^2}.$$

Since $9C - 4 = \frac{Z+4S}{2Z-S}$, $9C - 4 > 0$. Then, $Cost > K$ and we obtain a contradiction. Therefore $\sum_{i\in I} a_i = \frac{S}{2}$, and $\mathcal{I}_1$ has a solution, which concludes the proof.

■

Greedy heuristic for the *multi-stream* case

Since AND-Multi-Decision is NP-complete we propose a greedy scheduling heuristics, MultiStreamGreedy (Algorithm 9), which extends the ideas of SingleStreamGreedy to the *multi-stream* case.

SingleStreamGreedy computes a schedule by concatenating sequences of leaves. Each such sequence consists of leaves that all require data items from the same stream. These leaves are ordered by non-decreasing number of required data items. We extend this approach to the *multi-stream* case thanks to a notion of *dominance*. We say that leaf l_i dominates leaf l_j if for each stream s leaf l_i requires at least as many data items from s as l_j ($d^s_{l_i} \geq d^s_{l_j}$). MultiStreamGreedy considers all sequences of yet-to-be-scheduled leaves such that the $(i+1)$-th leaf in the sequence dominates the i-th leaf. Like SingleStreamGreedy, MultiStreamGreedy picks the sequence that has the lowest ratio of cost to probability of failure.

Consider an AND tree with a set of leaves $\mathcal{L} = l_1, \ldots, l_L$. MultiStreamGreedy takes $\mathcal{L}$ as input and returns a schedule, ξ, and its expected cost. While there remain leaves to schedule (while loop at line 6), MultiStreamGreedy computes all dominance relationships among the yet to be scheduled leaves via a call to DirectDomination.

DirectDomination(Algorithm 7) computes the direct (non-transitive) dominance relationships between the leaves. It calls TransitiveDomination(Algorithm 8), which computes the transitive dominance relationships.

Algorithm 7: DirectDomination($\mathcal{L}$, *NotYetScheduled*)

```
1  ($Source$, $DominatesTrans$) = TransitiveDomination($\mathcal{L}$, $NotYetScheduled$);
2  for $l_{first} \in \mathcal{L}$ do
3      for $l_{second} \in \{l_{first+1}, \dots, l_L\}$ do
4          $DominatesDirect[l_{first}][l_{second}] \leftarrow$ FALSE;
5  for $l_{first} \in \mathcal{L}$ do Check whether $l_{first}$ dominates directly $l_{second}$
6      if ($NotYetScheduled[l_{first}]$) then
7          for $l_{second} \in \mathcal{L} \setminus \{l_{first}\}$ do
8              if ($NotYetScheduled[l_{second}]$ and $DominatesDirect[l_{first}][l_{second}]$) then
9                  $NoMiddleLeaf \leftarrow$ TRUE;
10                 for $l_{middle} \in \mathcal{L} \setminus \{l_{first}, l_{second}\}$ do
11                     if $NotYetScheduled[l_{middle}]$
12                         and $DominatesDirect[l_{middle}][l_{second}]$
13                         and $DominatesDirect[l_{first}][l_{middle}]$) then
14                         $NoMiddleLeaf \leftarrow$ FALSE;
15                 $DominatesDirect[l_{first}][l_{second}] \leftarrow NoMiddleLeaf$;
16 return $DominatesDirect$;
```

DirectDomination returns the set of source leaves, i.e., leaves that dominate no other leaves (boolean array *Source*), and the set of dominance relationships (boolean array *Dominates*). These relationships are direct, i.e., not including transitive dominances. For each source leaf l_i, MultiStreamGreedy then examines all possible sequences starting with l_i (for loop at line 11). This is done via a call to the recursive GreedyKernel function (Algorithm 10), which computes the sequence that starts with l_i that provides the best extension to the current schedule. MultiStreamGreedy then selects the best extension among all these extensions for all source leaves (lines 14-18).

The complexity of GreedyKernel is $O(2^L)$. This is because GreedyKernel explores all paths starting from a given source leaf in the dominance relationship graph (in a directed acyclic graph with n vertices there are at most $O(2^n)$ paths). MultiStreamGreedy has complexity $O(L^2 2^L)$. This is because at each step MultiStreamGreedy schedules at least one leaf and calls GreedyKernel for each unscheduled source leaf. In spite of its exponential worst-case complexity, for the problem instances used in our experimental evaluations we are able to execute GreedyKernel in at most 0.03 sec (for a AND node with 40 leaves) on one core of an 2.1 GHz AMD Opteron processor.

Figure 7.6 shows results for a set of randomly generated AND trees. Instances are generated using the same method as that described in Section 7.4.2. For a given number of leaves $L = 2, \dots, 10$ and a given sharing ratio $\rho = 1, 5/4, 4/3, 3/2, 2, 3, 4, 5, 10$, we generate 1,000 random trees for a total of 81,000 random trees. The number of streams referenced by each leaf is sampled from a uniform distribution over the interval $[1, 5]$, and each such stream is sampled uniformly from the set of streams. For each tree we compute the cost achieved by MultiStreamGreedy and that achieved by a high-complexity exhaustive search for the optimal schedule (of complexity $O(L!)$). Figure 7.6 plots these costs for all instances, sorted by increasing optimal cost. The average, resp. maximum, relative difference between the results of MultiStreamGreedy and the results of the optimal algorithm is 0.60%, resp. 28.53%. The relative difference is larger than 5% for only 3.73% of the instances, and the two algorithms

Algorithm 8: TransitiveDomination($\mathcal{L}$, *NotYetScheduled*)

```
1  for $l_i \in \mathcal{L}$ do Initialization
2      $Source[l_i] \leftarrow$ TRUE
3  for $l_{first}$ in $\mathcal{L}$ do Computation of the dominance relationship
4      if ($NotYetScheduled[l_{first}]$) then
5          for $l_{second} \in \{l_{first+1}, \dots, l_L\}$ do
6              if ($NotYetScheduled[l_j]$) then
7                  $FirstDominatesSecond \leftarrow$ TRUE; $SecondDominatesFirst \leftarrow$ TRUE;
8                  for $s \in \mathcal{S}$ do Loop on all streams
9                      if ($d^s_{l_{first}} > d^s_{l_{second}}$) then
10                         $SecondDominatesFirst \leftarrow$ FALSE;
11                     else if ($d^s_{l_{first}} < d^s_{l_{second}}$) then
12                         $FirstDominatesSecond \leftarrow$ FALSE;
13                 if ($SecondDominatesFirst$ and $FirstDominatesSecond$) then
14                     if ($p_{l_{first}} < p_{l_{second}}$ ) then
15                         $DominatesTrans[l_{second}][l_{first}] \leftarrow$ TRUE;
16                         $DominatesTrans[l_{first}][l_{second}] \leftarrow$ FALSE;
17                         $Source_{l_{second}} \leftarrow$ FALSE;
18                     else
19                         $DominatesTrans[l_{second}][l_{first}] \leftarrow$ FALSE;
20                         $DominatesTrans[l_{first}][l_{second}] \leftarrow$ TRUE;
21                         $Source_{l_{first}} \leftarrow$ FALSE;
22                 else
23                     $DominatesTrans[l_{first}][l_{second}] \leftarrow FirstDominatesSecond$;
24                     $DominatesTrans[l_{second}][l_{first}] \leftarrow SecondDominatesFirst$ ;
25                     if ($FirstDominatesSecond$) then
26                         $Source[l_{first}] \leftarrow$ FALSE;
27                     else if ($SecondDominatesFirst$) then
28                         $Source[l_{second}] \leftarrow$ FALSE;
29 return ($Source$, $DominatesTrans$);
```

lead to the same cost for 76.75% of the instances. We conclude that MultiStreamGreedy is likely to provide close-to-optimal schedules in *multi-stream* case for *shared* queries.

Algorithm 9: MultiStreamGreedy($\mathcal{L}$)

```
for $l_i \in \mathcal{L}$ do
    $NotYetScheduled[l_i] \leftarrow$ TRUE;
$NumNotYetScheduled \leftarrow |\mathcal{L}|$;
$ScheduleCost \leftarrow 0$;
$\xi \leftarrow \varnothing$;
while $(NumNotYetScheduled > 0)$ do
    $(Source, Dominates) =$ DirectDomination($\mathcal{L}$, $NotYetScheduled$);
    $MinRatio \leftarrow +\infty$;
    $BestExt \leftarrow \varnothing$;
    $CostBestExtension \leftarrow +\infty$;
    for $l_i \in \mathcal{L}$ do
        if $(NotYetScheduled[l_i]$ and $Source[l_i])$ then
            $(Cost, Extension, Proba) =$
                GreedyKernel($\mathcal{L}, \xi, ScheduleCost, 1, l_i, Dominates$);
            $Ratio \leftarrow (Cost - ScheduleCost)/(1 - Proba)$;
            if $(Ratio < MinRatio)$ then
                $MinRatio \leftarrow Ratio$;
                $BestExt \leftarrow Extension$;
                $CostBestExtension \leftarrow Cost$;
    $\xi \leftarrow \xi \cdot BestExt$;
    $ScheduleCost \leftarrow CostBestExtension$;
    for $l_i \in BestExt$ do
        $NotYetScheduled[l_i] \leftarrow$ FALSE;
        $NumNotYetScheduled \leftarrow NumNotYetScheduled - 1$;
return $(\xi, ScheduleCost)$
```

7.5 DNF trees

In this section we focus on DNF trees for *shared* queries. We first show that, as for *read-once* queries, there is an optimal schedule that is depth-first (Section 7.5.1). This result holds in the *multi-stream* case, and thus in the less general *single-stream* case. We then show that the problem is NP-complete in the *single-stream* case (Section 7.5.2), and thus also NP-complete in the more general *multi-stream* case. We then propose and evaluate several heuristics (Section 7.5.3).

7.5.1 Dominance of depth-first schedules

Theorem 11. *Given a* DNF *tree, there exists an optimal schedule that is depth-first, i.e., that processes* AND *nodes one by one.*

Proof. Consider a DNF tree $\mathcal{T}$ and a schedule ξ. We use the same notations as in Section 7.3. Without loss of generality we assume that the AND nodes, $\text{AND}_1, \ldots, \text{AND}_N$, are numbered in the order of their completion in ξ. Thus, according to ξ, AND_1 is the first AND node

Algorithm 10: GreedyKernel($\mathcal{L}, \xi, BaseCost, BaseProba, Leaf, Dominates$)

```
1  $CostBestSol \leftarrow \textsc{Cost}(\xi \cdot Leaf)$;
2  $BestExt \leftarrow Leaf$;
3  $ProbaBestExt \leftarrow p_{Leaf}$;
4  $BestRatio \leftarrow (CostBestSol - BaseCost)/(1 - BaseProba \times p_{Leaf})$;
5  for $l_j \in \mathcal{L}$ do
6      if $Dominates[l_j][Leaf]$ then
7          $(Cost, Ext, Proba) \leftarrow$
           $\textsc{GreedyKernel}(\mathcal{L}, \xi \cdot Leaf, BaseCost, BaseProba \times p_{Leaf}, l_j, Dominates)$;
8          $Ratio \leftarrow (Cost - BaseCost)/(1 - BaseProba \times p_{Leaf} \times Proba)$;
9          if $(Ratio < BestRatio)$ then
10             $BestRatio \leftarrow Ratio$;
11             $CostBestSol \leftarrow Cost$;
12             $ProbaBestExt \leftarrow p_{Leaf} \times Proba$;
13             $BestExt \leftarrow Leaf \cdot Ext$;
14 return $(CostBestSol, BestExt, ProbaBestExt)$
```

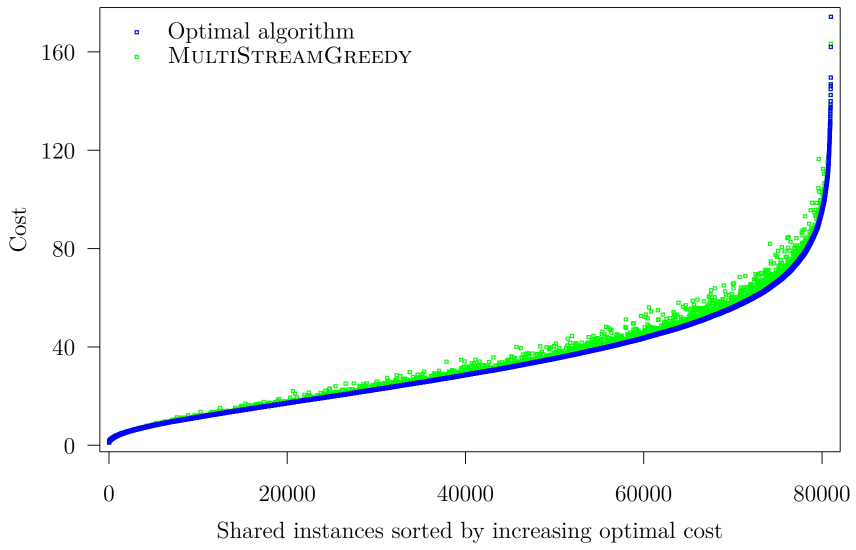


Figure 7.6: Cost achieved by MultiStreamGreedy and that achieved by the optimal algorithm, shown for 81,000 random AND tree instances sorted by increasing optimal cost.

with all its leaves evaluated. We denote by K the number (possibly zero) of AND nodes that are processed one by one and entirely at the beginning of the query processing according to ξ. Therefore, if ξ evaluates a leaf $l_{i,j}$, with $i \neq 1$, in the m_1 first steps, then $K = 0$. Finally, we assume that the leaves of each AND node are numbered according to their evaluation order in ξ.

We prove the theorem by contradiction. Let us assume that there does not exist an optimal schedule with $K = N$. Let ξ be an optimal schedule that maximizes K. By definition of K and by the hypothesis on the numbering of the AND nodes, schedule ξ evaluates some leaves of the AND nodes AND$_{K+2}$, ..., AND$_N$ before it evaluates the last leaf of AND$_{K+1}$. Let $\hat{\mathcal{L}}$ denote the set of these leaves. We now define a new schedule ξ' that starts by executing at least $K+1$ AND nodes one by one:

— ξ' starts by evaluating the first K AND nodes one by one, evaluating their leaves in the same order and at the same steps as in ξ;
— ξ' then evaluates all the leaves of AND$_{K+1}$ in the same order as in ξ (but not at the same steps);
— ξ' then evaluates the leaves in $\hat{\mathcal{L}}$ in the same order as in ξ (but not at the same steps);
— ξ' finally evaluates the remaining leaves in the same order and at the same steps as in ξ.

The cost of a schedule is the sum, over all potentially acquired data items, of the cost of acquiring each data item times the probability of acquiring it. Let d be a data item potentially needed by a leaf in $\mathcal{T}$. We show that the probability of acquiring d is not greater with ξ' than with ξ. We have three cases to consider.

Case 1) d is not needed by any leaf of AND$_{K+1}$ and not needed by any leaf in $\hat{\mathcal{L}}$. Then d's probability to be acquired is the same with ξ and ξ'.

Case 2) d is needed by at least one leaf of AND$_{K+1}$. The only way in which a leaf that is evaluated in ξ would not be evaluated in ξ' is if AND$_{K+1}$ evaluates to TRUE. Since at least one leaf of AND$_{K+1}$ uses d, for AND$_{K+1}$ to evaluate to TRUE d must be acquired. Consequently, the probability that d is acquired is the same with ξ and with ξ'.

Case 3) d is needed by at least one leaf in $\hat{\mathcal{L}}$ but not needed by any leaf of AND$_{K+1}$. ξ and ξ' define the same ordering on the leaves in $\hat{\mathcal{L}}$. For each AND node AND$_i$, with $K+2 \leq i \leq N$, there is at most one leaf in AND$_i \cap \hat{\mathcal{L}}$ that can be the leaf responsible for the acquisition of d with ξ, and it is the same leaf with ξ'. Let $\mathcal{F}$ be the set of all these leaves. Then, with ξ, the leaves in $\mathcal{F}$ are responsible for the acquisition of d if and only if:

— AND$_1$, ..., AND$_K$ all evaluate to FALSE;
— None of the evaluated leaves of AND$_1$, ..., AND$_K$ needs d; and
— At least one of the leaves in $\mathcal{F}$ is evaluated.

Let us denote by $\mathcal{P}$ the probability that all the AND nodes AND$_1$, ..., AND$_K$ evaluate to FALSE and that none of the evaluated leaves of these AND nodes needs the data item d. Let us denote by $\mathcal{D}$ the probability that d is acquired because of the evaluation of one of the leaves of the AND nodes AND$_1$, ..., AND$_K$. Finally, let $\mathcal{R}$ be the probability that one of the leaves evaluated with ξ after $l_{K+1,m_{K+1}}$ acquires d, knowing that no leaves of AND$_1$, ..., AND$_K$ or in $\hat{\mathcal{L}}$ acquires it. Then, with ξ, the probability p that d is acquired is:

$$p = \mathcal{D} + \mathcal{P}\left(1 - \prod_{l_{i,j}\in\mathcal{F}}\left(1 - \prod_{k=1}^{j-1} p_{i,k}\right)\right) + \mathcal{R} \qquad (7.10)$$

because leaf $l_{i,j}$ is evaluated with probability $\prod_{k=1}^{j-1} p_{i,k}$, that is, if all the leaves from the same

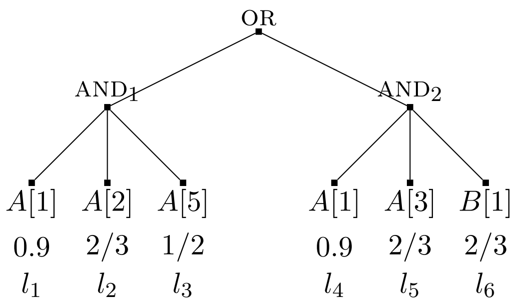


Figure 7.7: Example DNF tree for which SingleStreamGreedy does not produce an optimal schedule.

AND node that are evaluated prior to it all evaluate to TRUE. The second term of Equation (7.10) is the probability that the leaves in $\mathcal{F}$ are responsible for acquiring d.

With schedule ξ', the leaves of $\mathcal{F}$ are responsible for the acquisition of d if and only if:
— The AND nodes AND$_1$, ..., AND$_K$, and AND$_{K+1}$ all evaluate to FALSE;
— None of the evaluated leaves of the AND nodes AND$_1$, ..., AND$_K$ need d; and
— At least one of the leaves in $\mathcal{F}$ is evaluated.
Thus, with ξ', the probability p' that d is acquired is:

$$p' \quad = \quad \mathcal{D} \; + \; \mathcal{P}\left(1-\prod_{k=1}^{m_{K+1}} p_{K+1,k}\right) \; \times \; \left(1-\prod_{l_{i,j}\in\mathcal{F}}\left(1-\prod_{k=1}^{j-1} p_{i,k}\right)\right) \; + \; \mathcal{R}.$$

Comparing this equation with Equation 7.10, we see that p' is not greater than p. The probability that a data item is acquired with ξ' is thus not greater than with ξ. Therefore, in each of the three cases the cost of ξ' is not greater than the cost of ξ, meaning that ξ' is also an optimal schedule. Since ξ' starts by executing at least $(K+1)$ AND nodes one by one, we obtain a contradiction with the maximality assumption on K, which concludes the proof. ■

7.5.2 The *single-stream* case is NP-complete

For *read-once* queries an optimal algorithm for DNF trees is built on top of the optimal algorithm for AND trees in [105]. The same approach cannot be used for *shared* queries (i.e., reusing SingleStreamGreedy). This can be shown by a simple counter-example in the following paragraph.

In other words, for some DNF trees, the ordering of the leaves of a given AND node in an optimal schedule does not correspond to the ordering produced by SingleStreamGreedy for that AND node. In fact, we show that finding an optimal schedule for evaluating a DNF tree is NP-complete.

Counter-example for Algorithm 6 on DNF trees We provide a counterexample to show that in the *shared* case, SingleStreamGreedy (optimal to evaluate a AND tree) cannot be used to evaluate a DNF tree.

We consider the example of Figure 7.7 where both streams have a cost of 1. There are two possible orders for evaluating the AND nodes. We consider both of them and explicit the behavior of SingleStreamGreedy on each of the AND's.

— AND$_1$ then AND$_2$. Because of Proposition 15, the leaves of AND$_1$ are always evaluated in the order l_1, l_2, l_3. The cost of evaluation of AND$_1$ is then $\alpha_1 = 1+0.9\times(1+2/3\times3) = 3.7$. We then move to the evaluation of AND$_2$. SingleStreamGreedy first schedules leaf l_5 whose cost is null. We then have to compare the ratios for leaves l_5 and l_6:
  — l_5. The first element of A was acquired for the evaluation of leaf l_1. The second element of A needs to be acquired for l_2 only if leaf l_2 was not evaluated and leaf l_4 evaluated to TRUE, which happens with probability $(1-0.9)\times0.9 = 0.09$. The third element of A needs to be acquired only if leaf l_3 was not evaluated and leaf l_4 evaluated to TRUE, which happens with probability $(1-0.9\times2/3)\times0.9 = 0.36$. Therefore, the evaluation cost of l_5 is $0+0.09\times1+0.36\times1 = 0.45$. The ratio for leaf l_5 is thus $\frac{0.45}{1-2/3} = 1.35$.
  — l_6. The ratio for l_6 is $\frac{1}{1-2/3} = 3$.

Therefore, SingleStreamGreedy schedules l_5 and then l_6 for the overall cost:

$$3.7+0+0.45+(1-0.9\times2/3\times1/2)\times0.9\times2/3 = 4.57.$$

— AND$_2$ then AND$_1$. We first consider the ratios for the three leaves:
  — The ratio for l_4 is $\frac{1}{1-0.9} = 10$.
  — The ratio for l_5 is $\frac{1+0.9\times2}{1-0.9\times2/3} = 7$.
  — The ratio for l_6 is $\frac{1}{1-2/3} = 3$.

Therefore the first scheduled leaf if l_6. Then the overall schedule is l_6, l_4, l_5, l_1, l_2, l_3. We compute the evaluation cost of each leaf.
  — l_6. Its cost is 1.
  — l_4. Its cost is $2/3\times1$.
  — l_5. Its cost is $2/3\times0.9\times2 = 1.2$.
  — l_1. Its cost is $(1-2/3)\times1$.
  — l_2. Its costs is $(1-2/3\times0.9)\times0.9 = 0.36$.
  — l_3. Its costs is $(1-2/3\times0.9)\times1+(1-2/3\times0.9\times2/3)\times0.9\times2/3\times2 = 0.96$.

The overall cost is thus 4.52.

We now consider the schedule l_4, l_5, l_6, l_1, l_2, l_3. We compute the evaluation cost of each leaf.
— The cost of l_4 is 1.
— The cost of l_5 is $0.9\times2 = 1.8$.
— The cost of l_6 is $0.9\times2/3\times1 = 0.6$.
— The cost of l_1 is 0.
— The cost of l_2 is $(1-0.9)\times0.9 = 0.09$.
— The cost of l_3 is $(1-0.9)\times0.9\times2/3+(1-0.9\times2/3\times2/3)\times0.9\times2/3\times2 = 0.78$.

The overall cost of this schedule is thus 4.27. The intuitive explanation is as follows: for AND$_2$ alone, the best evaluation order is l_6, l_4, l_5 (and this is the order chosen by SingleStream-Greedy). However, because of the re-use of some data items of stream A in AND$_1$, the optimal order for the whole DNF tree is not the same! This leads to a enormous combinatorial search space for the optimal ordering, which corroborates the hardness result (NP-completeness stated in Theorem 12) of the evaluation of DNF trees in the *shared* case.

Definition 2 (DNF-Single-Decision)**.** Given a *single-stream* DNF tree and a cost bound K, is there a schedule whose expected cost does not exceed K?

Theorem 12. DNF-SINGLE-DECISION *is NP-complete.*

Proof. The problem is obviously in NP: given a schedule, i.e., an ordering of the leaves, one can compute its expected cost in polynomial time, using the method given in Section 7.3.2. The NP-completeness is obtained by reduction from 2-PARTITION [94]. Let $\mathcal{I}_1$ be an instance from 2-PARTITION: given a set $\{a_1, ..., a_n\}$ and $S = \sum_{i=1}^{n} a_i$, does there exist a subset I such that $\sum_{i\in I} a_i = \frac{S}{2}$? We assume that S is even, otherwise there is no solution. The size of $\mathcal{I}_1$ is $O(n \log M)$, where $M = \max_{1\leq i\leq n}\{a_i\}$. Without loss of generality, we assume that $M \geq 10$. We construct the following instance $\mathcal{I}_2$ of DNF-DECISION:

— We consider a DNF tree with $N = n + 1$ AND nodes AND$_i$, $1 \leq i \leq n + 1$ and a total of $L = 2n + 1$ leaves.

— The set of streams is $\mathcal{S} = \{A_1, \ldots, A_n, B\}$. The cost of stream $s_i = A_i$ for $i \leq n$ is $c(i) = \frac{1}{2Z}$, where Z is some large constant defined below. The cost of stream $s_{n+1} = B$ is $c(n + 1) = C_0$, where $C_0 \approx \frac{1}{2}$ is a constant defined below.

— Each AND$_i$ node, where $i \leq n$, has a single leaf $l_{i,1}$ which has success probability

$$p_{i,1} = \frac{a_i}{Z} + \beta\frac{a_i^2}{Z^2}$$

where $\beta \approx \frac{1}{2}$ is a constant defined below, and which requires $d_{l_{i,1}}^{A_i}$ elements of stream A_i. Hence the cost to access all items of leaf $l_{i,1}$ is $d_{l_{i,1}}^{A_i} c(i) = \frac{a_i}{Z}$.

— The last AND node AND$_{n+1}$ has $m_{n+1} = n + 1$ leaves which are specified as follows:

  — Each leaf $l_{n+1,i}$, where $i \leq n$, has success probability $p_{n+1,i} = 1 - \varepsilon$ and requires $d_{l_{n+1,i}}^{A_i} = a_i$ elements of stream A_i. Hence the cost to access all items of leaf $l_{n+1,i}$ is $\frac{a_i}{2Z}$.

  — The last leaf $l_{n+1,n+1}$ has success probability $p_{n+1,n+1} = 1-\varepsilon$ and requires $d_{l_{n+1,n+1}}^{B} = 1$ element of stream B (at cost $c(n+1) = C_0$). Constant ε is chosen to be very small, see below. Let $C = \sum_{i=1}^{n} \frac{a_i}{2Z} + C_0 = \frac{S}{2Z} + C_0$: Intuitively, C would be the cost of evaluating node AND$_{n+1}$ when starting with this AND node, and when ε becomes negligible.

— The bound on the expected evaluation cost is $K = C\left(1 - \frac{S^2}{8Z^2}\right) + \frac{1}{9Z^2}$

To finalize the description of $\mathcal{I}_2$, we define the constants as follows:

— $Z = 10\left((n+1)3^n + n^3\right)M^3$

— $C_0 = \frac{Z}{2Z-S} - \frac{S}{2Z}$, so that $C = \frac{Z}{2Z-S}$

— $\beta = \frac{1-C}{2C}$

— $\varepsilon = \frac{1}{90(n+1)^2Z^2}$

The size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$: the greatest value in $\mathcal{I}_2$ is Z and $\log(Z)$ is linear in $(n + \log M)$. Because Z is very large in front of $S \leq nM$, we do have that C, C_0 and β are all close to $\frac{1}{2}$. We only use that these constants are all non-negative, and that $\beta \leq 1$ and $C \leq 1$, in the following derivation, where we bound the expected cost of an arbitrary evaluation of the DNF tree. Then, using this derivation, we will prove that $\mathcal{I}_1$ has a solution I if and only if $\mathcal{I}_2$ does.

Let us start with the cost of an arbitrary evaluation of the DNF tree. Owing to the dominance property stated in Theorem 11, we can assume that the schedule is depth-first. Therefore, a schedule first evaluates n AND nodes in sequence and completely before starting the evaluation of node AND$_{n+1}$. Then, because ε is very small, we can compute an approximation of the cost by assuming that the schedule terminates after node AND$_{n+1}$. This is because all its leaves have success probability close to 1. We will bound the difference between this approximation and the actual cost later on.

Let $I = \{\mathrm{AND}_{\sigma(1)}, \mathrm{AND}_{\sigma(2)}, \ldots, \mathrm{AND}_{\sigma(k)}\}$ be the subset, of cardinal k, of AND nodes that are evaluated, in that order, before node AND_{n+1}. Let C be the approximated cost of the schedule (terminating after completion of node AND_{n+1}). To simplify notations, we let $x_i = a_{\sigma(i)}$ for $1 \leq i \leq k$, and let $q_i = 1 - p_{\sigma(i),1}$ for $i \leq n$. By definition,

$$C = \sum_{i=1}^{k} \frac{x_i}{Z} \prod_{1\leq j<i} q_j + \left(C - \sum_{i=1}^{k} \frac{x_i}{2Z}\right) \prod_{1\leq j\leq k} q_j.$$

Note that the cost of node AND_{n+1} has been reduced from its original value, due to the sharing of the streams whose index is in I. To evaluate C, we start by approximating

$$\prod_{1\leq j<i} q_j = \prod_{1\leq j<i} \left(1 - \frac{x_j}{Z} - \beta\frac{x_j^2}{Z^2}\right)$$

Let

$$F_i = 1 - \sum_{j=1}^{i-1} \frac{x_j}{Z} - \beta \sum_{j=1}^{i-1} \frac{x_j^2}{Z^2} + \sum_{1\leq j_1<j_2<i} \frac{x_{j_1}x_{j_2}}{Z^2}.$$

We have

$$\left|\left(\prod_{1\leq j<i} q_j\right) - F_i\right| \leq \frac{3^n M^3}{Z^3} \tag{7.11}$$

To see this, we have kept in F_i all terms of the product $\prod_{1\leq j<i} q_j$ whose denominators include a factor strictly inferior to Z^3. The other terms of the product are bounded (in absolute value) by M^3/Z^3, because $\beta \leq 1$ and $M \leq Z$. There are at most $3^{i-1} \leq 3^n$ such terms. Hence, the desired bound in Equation (7.11). Letting

$$G = \sum_{i=1}^{k} \frac{x_i}{Z} - \sum_{1\leq j_1<j_2\leq k} \frac{x_{j_1}x_{j_2}}{Z^2},$$

we prove similarly that

$$\left|\left(\sum_{i=1}^{k} \frac{x_i}{Z} \prod_{1\leq j<i} q_j\right) - G\right| \leq \frac{n3^n M^3}{Z^3}. \tag{7.12}$$

Indeed, there are $k \leq n$ terms in the sum, each of them being bounded as before. We deduce from Equations (7.11) and (7.12), using $C \leq 1$, that

$$\left|C - \left(G + (C - \sum_{i=1}^{k} \frac{x_i}{2Z})F_{k+1}\right)\right| \leq \frac{(n+1)3^n M^3}{Z^3} \tag{7.13}$$

Now, we aim at simplifying $H = G + (C - \sum_{i=1}^{k} \frac{x_i}{2Z})F_{k+1}$ by dropping terms whose denominator is Z^3. We have

$$H = \sum_{i=1}^{k} \frac{x_i}{Z} - \sum_{1\leq j_1<j_2\leq k} \frac{x_{j_1}x_{j_2}}{Z^2} + \left(C - \sum_{i=1}^{k} \frac{x_i}{2Z}\right)\left(1 - \sum_{j=1}^{k} \frac{x_j}{Z} - \beta \sum_{j=1}^{k} \frac{x_j^2}{Z^2} + \sum_{1\leq j_1<j_2\leq k} \frac{x_{j_1}x_{j_2}}{Z^2}\right)$$

Defining

$$\tilde{H} = C + \frac{1-2C}{2Z}\sum_{i=1}^{k} x_i + \frac{1}{2Z^2}\left(\sum_{i=1}^{k} x_i\right)^2 + \frac{C-1}{Z^2}\sum_{1\le j_1<j_2\le k} x_{j_1}x_{j_2} - \frac{\beta C}{Z^2}\sum_{i=1}^{k} x_i^2,$$

we derive (using $\beta \le 1$) that:

$$\left|H-\tilde{H}\right| \le \left|\frac{1}{2Z^3}\left(\sum_{i=1}^{k} x_i\right)\left(\sum_{1\le j_1<j_2\le k} x_{j_1}x_{j_2} + \sum_{i=1}^{k} x_i^2\right)\right|.$$

Hence,

$$\left|H-\tilde{H}\right| \le \frac{n^3M^3}{Z^3} \tag{7.14}$$

Developing $(\sum_{i=1}^{k} x_i)^2 = \sum_{i=1}^{k} x_i^2 + 2\sum_{1\le j_1<j_2\le k} x_{j_1}x_{j_2}$ in $\tilde{H}$, we obtain

$$\tilde{H} = C + \frac{1-2C}{2Z}\sum_{i=1}^{k} x_i + \frac{C}{Z^2}\sum_{1\le j_1<j_2\le k} x_{j_1}x_{j_2} + \frac{1-2\beta C}{2Z^2}\sum_{i=1}^{k} x_i^2$$

We have chosen the constants C and β so that $\tilde{H}$ can be reduced to

$$\tilde{H} = C + \frac{C}{2Z^2}\left(\left(\frac{S}{2} - \sum_{i=1}^{k} x_i\right)^2 - \frac{S^2}{4}\right) \tag{7.15}$$

Indeed, we have $\frac{1-2C}{2Z} = \frac{-SC}{2Z^2}$, and $C = 1-2C\beta$. Altogether, we derive from Equations (7.13) to (7.15) that

$$\left|C - C\left(1-\frac{S^2}{8Z^2}\right) - \frac{C}{2Z^2}\left(\frac{S}{2} - \sum_{i=1}^{k} x_i\right)^2\right| \le \frac{\left((n+1)3^n + n^3\right)M^3}{Z^3} = \frac{1}{10Z^2} \tag{7.16}$$

Finally, we coarsely bound the difference between the actual cost $Cost$ of the schedule and the approximated cost C. The actual probability of evaluating the i-th leaf of node AND$_{n+1}$ is $(1-\varepsilon)^i$ so that the error term for that leaf does not exceed $(1-(1-\varepsilon)^i)\max(\frac{M}{2Z}, C) \le n\varepsilon$. Since there are $n+1$ terms, we get a difference bounded by $n(n+1)\varepsilon$. Next we have neglected the evaluation of the remaining AND nodes after node AND$_{n+1}$, but this cost is (similarly) bounded by $(n+1)\varepsilon\frac{S}{Z} \le (n+1)\varepsilon$. Altogether, we obtain that

$$|Cost - C| \le (n+1)^2\varepsilon = \frac{1}{90Z^2} \tag{7.17}$$

Combining Equations (7.16) and (7.17), we finally derive that

$$\left|Cost - C\left(1-\frac{S^2}{8Z^2}\right) - \frac{C}{2Z^2}\left(\frac{S}{2} - \sum_{i=1}^{k} x_i\right)^2\right| \le \frac{1}{9Z^2} \tag{7.18}$$

We now prove that $\mathcal{I}_1$ has a solution I if and only if $\mathcal{I}_2$ does. Suppose first that $\mathcal{I}_1$ has a solution I: $\sum_{i\in I} a_i = \frac{S}{2}$. We evaluate the AND nodes whose indices are in I before evaluating node AND_{n+1}, followed by the remaining AND nodes in any order. Let *Cost* be the cost of this evaluation. From Equation (7.18), we have

$$\left| Cost - C\left(1 - \frac{S^2}{8Z^2}\right)\right| \leq \frac{1}{9Z^2}.$$

Hence, $Cost \leq C\left(1 - \frac{S^2}{8Z^2}\right) + \frac{1}{9Z^2} = K$, thereby providing a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution whose cost is $Cost \leq K$, and let I denote the (index) set of AND nodes that are evaluated before node AND_{n+1}. If (by contradiction) we have $\sum_{i\in I} a_i \neq \frac{S}{2}$, then $\left(\frac{S}{2} - \sum_{i=1}^{k} x_i\right)^2 \geq 1$, and Equation (7.18) shows that

$$Cost \geq C\left(1 - \frac{S^2}{8Z^2}\right) + \frac{C}{2Z^2} - \frac{1}{9Z^2} = K + \frac{9C-4}{9Z^2}.$$

Since $9C - 4 = \frac{Z+4S}{2Z-S} > 0$, then $Cost > K$, which is a contradiction. Therefore $\sum_{i\in I} a_i = \frac{S}{2}$, and $\mathcal{I}_1$ has a solution. This concludes the proof.

It is interesting to point out that instance $\mathcal{I}_2$ is constructed so that the ordering of the leaves inside each AND node has no importance. In fact, only the last AND node has more than one leaf, and because its leaves have all very high success probability, their ordering does not matter. This shows that the combinatorial difficulty of the DNF-Decision problem already lies in deciding the ordering of the AND nodes. ■

It is interesting to point out that in the above proof instance $\mathcal{I}_2$ is constructed so that the ordering of the leaves inside each AND node has no importance. In fact, only the last AND node has more than one leaf, and because its leaves have all high success probability, their ordering does not matter. This shows that the combinatorial difficulty of the DNF-Decision problem already lies in deciding the ordering of the AND nodes. However, even if the optimal order of the AND nodes is given, an optimal schedule cannot be computed by simply using SingleStreamGreedy (which is optimal for a single AND node) for scheduling the leaves of each AND node. See a counter-example in 7.5.2.

7.5.3 Heuristics and Evaluation Results

Given the NP-completeness result in the previous section we propose several heuristics. Most of these heuristics apply to both the *single-stream* and the *multi-stream* cases, one heuristic applies only to the *single-stream* case, and another applies only to the *multi-stream* case, as described in the following sections.

Heuristics common to the *single-stream* and *multi-stream* cases

We propose two categories of heuristics, which we term leaf-ordered and AND-ordered. *Leaf-ordered heuristics* simply sort the leaves according to costs ($\mathcal{C}$), failure probabilities ($q = 1 - p$), or the ratio of the two, which leads to three heuristics plus a baseline random one:

— Leaf-ordered, non-increasing q (prioritizes leaves with high chances of shortcutting the evaluation of an AND node);

— Leaf-ordered, non-decreasing $\mathcal{C}$ (prioritizes leaves with low costs);

— Leaf-ordered, non-decreasing $\mathcal{C}/q$ (prioritizes leaves with low costs and also with high chances of shortcutting the evaluation of an AND node);

— Leaf-ordered, random (baseline).

The above first three heuristics have intuitive rationales. Other options are possible (e.g., sort leaves by non-increasing $\mathcal{C}$) but are easily shown to produce poor results.

AND-*ordered heuristics*, unlike leaf-ordered heuristics, account for the structure of the DNF tree by building depth-first schedules, with the rationale that there is a depth-first schedule that is optimal (Theorem 11). The heuristics thus proceed in two phases: (i) compute a schedule for the leaves of each AND node independently, ignoring the other AND nodes, using SingleStreamGreedy (optimal) in the *single-stream* case and MultiStreamGreedy (sub-optimal) in the *multi-stream* case; (ii) pick an order of the AND nodes and concatenate their individual leaf schedules. Given the individual schedule of each AND node computed in the first phase, we can compute the (expected) cost and the probability of success of that AND node (using the method in Section 7.3). In the second phase, the AND-ordered heuristics simply order the AND nodes based on their computed costs ($\mathcal{C}$), computed probability of success (p), or ratio of the two, leading to three heuristics:

— AND-ordered, non-increasing p (prioritizes AND's with high chances of shortcircuiting the evaluation of the OR node);

— AND-ordered, non-decreasing $\mathcal{C}$ (prioritizes AND's with low costs);

— AND-ordered, non-decreasing $\mathcal{C}/p$ (prioritizes AND's with low costs and also with high chances of shortcircuiting the evaluation of the OR node);

There are two approaches to compute the cost of an AND node: (i) consider the AND node in isolation assuming that the OR node has a single AND node child; or (ii) account for previously scheduled AND nodes whose evaluation has caused some data items to be acquired with some probabilities. We terms the first approach "static" and the second approach "dynamic," giving us two versions of the last two heuristics above.

A greedy stream-ordered heuristic for the *single-stream* case

For the *single-stream* case we propose heuristic a heuristic that orders the streams for data acquisition. The idea of this heuristic was proposed in [100], and to the best of our knowledge it is the only previously proposed heuristic for solving the PAOTR problem for *shared* DNF query trees.

The *stream-ordered* heuristic proceeds by ordering the streams from which data items are acquired, acquiring all items from a stream before proceeding to the next stream, until the truth value of the OR node has been determined. For each stream s_s the heuristic computes a metric, $R(s_s)$, defined as follows:

$$R(s_s) = \frac{\sum_{i,j|d^{s_s}_{l_{i,j}}>0} q_{i,j} n_{i,j}}{\max_{i,j|d^{s}_{l_{i,j}}>0} d^{s_s}_{l_{i,j}} c(s_s)},$$

where $n_{i,j}$ is the number of leaves whose evaluation would be shortcircuited if leaf $l_{i,j}$ was to evaluate to FALSE. The numerator can thus be interpreted as the shortcutting power of stream s_s. The denominator is the maximum data element acquisition cost over all the leaves that use stream s_s. The heuristic orders the streams by non-decreasing R values. The rationale is that one should prioritize streams that can shortcut many leaf evaluations and that have low maximum data item acquisition costs. The heuristic as it is described in [100] acquires the maximum number of needed data items from each stream so as to compute truth values of all the leaves that require data items from that stream. In other words, the leaves that require data items from stream s are scheduled in decreasing $d^{s_s}_{l_{i,j}}$ order. However, Proposition 15 holds for DNF trees, showing that it is always better to schedule these leaves in *increasing* $d^{s_s}_{l_{i,j}}$ order. We use this leaf order to implement this heuristic in this work. We have verified in our experiments that this version outperforms the version in [100] in the vast majority of the cases, with all remaining cases being ties.

A dynamic programming heuristic for the *multi-stream* case

When faced with an intractable problem, one option is find a related problem for which an optimal solution can be computed. In this view we propose a dynamic programming heuristic, MultiStreamDP (Algorithm 11), which computes not an optimal order of the leaves, but an optimal order of the individual data item retrievals. The hope is that these two objectives are sufficiently related that the optimal data item retrieval order induces a good leaf schedule.

MultiStreamDP builds a data item retrieval order as follows. Let $Max[s]$ be the maximum number of data items from stream s_s required by any leaf. The algorithm constructs an S-dimensional array DPCost. DPCost$[n_1, \ldots, n_S]$ denotes the expectation of the cost to acquire all the potentially needed data items knowing that n_i data items have already been acquired from stream s_i for all $i = 1, \ldots, S$. The goal is to compute DPCost$[0, \ldots, 0]$. Algorithm 11 first computes $Max[s]$ for all s (lines 1-6). It then iteratively computes DPCost values by non-increasing total number of already acquired data item starting with DPCost$[Max[1], \ldots, Max[S]] = 0$ (lines 7-10). Each computation is accomplish by a call to DPK (Algorithm 12).

DPK begins (lines 1-13) by computing, given the already acquired data items, which leaves are already evaluated, which AND nodes are already fully evaluated, and which streams have data items required by AND nodes that are not fully evaluated. From lines 20 to 29, the algorithm computes for each stream s_s the expected cost of retrieving the next data item from s_s. This computation relies on the probability that all fully evaluated AND nodes evaluate to FALSE (which is computed at lines 14-17). It also relies on the probability that all non-fully evaluated AND nodes requiring at least one data item from s_s so far evaluate to FALSE (which is computed at lines 22-24). The algorithm computes the lowest expected cost when the next acquired data item is from stream s_s (line 26). The desired DPCost value is the lowest such cost.

DPK has complexity $O(SL)$. Let $D = \max_{i\in\{1,\ldots,S\}} Max[i]$. MultiStreamDP places $(D+1)^S$ calls to DPK, for an overall complexity of $O(D^S LS)$. This complexity is exponential in the number of streams, which may preclude the use of MultiStreamDP in practice for instance with more than a few streams.

From the output of MultiStreamDP we must construct a leaf schedule. Based on the data item retrieval order we compute a completion order of the AND nodes, and an evaluation order of the leaves within each AND node. We then construct a depth-first schedule according to these orders.

Evaluation results for the *single-stream* case

In total, we consider 4 leaf-ordered, 5 AND-ordered, and 1 stream-ordered heuristics. We first evaluate these heuristics on a set of “small” instances for which we can compute optimal schedules using an exponential-time algorithm that performs an exhaustive search. Such an algorithm is feasible because, thanks to Theorem 11, it only needs to search over all possible depth-first schedules. Instances are generated using the same method as that described in

Algorithm 11: MultiStreamDP $(\mathcal{T})$

```
1  for $s = 1$ to $S$ do
2      $Max[s] \leftarrow 0$;
3      for $c = 1$ to $N$ do
4          for $i = 1$ to $m_c$ do
5              if $Max[s] < d^{s_s}_{l_{c,i}}$ then
6                  $Max[s] \leftarrow d^{s_s}_{l_{c,i}}$
7  DPCost$[Max[1], ..., Max[S]] \leftarrow 0$;
8  for $step = \left(\sum_{i=1}^{S} Max[i]\right) - 1$ down to 1 do
9      foreach $(n_1, ..., n_S)$ such that $\sum_{i=1}^{S} n_i = step$, with $n_i \leq Max[i]$ for each $i$ do
10         DPK$(\mathcal{T}$, DPCost$, n_1, ..., n_S)$
```

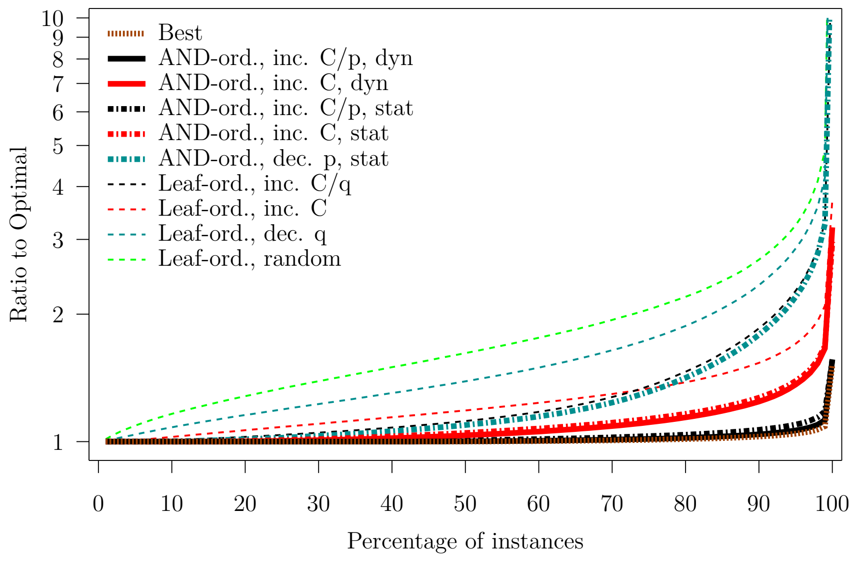


Figure 7.8: Ratio to optimal vs. fraction of the instances for which a smaller ratio is achieved, computed over 21,600 random "small" DNF tree instances in the *single-stream* case.

Algorithm 12: DPK($\mathcal{T}$, DPCOST, $n_1, ..., n_S$)

```
1  for $c = 1$ to $N$ do
2      $AndCompleted[c] \leftarrow$ TRUE;
3      $ProbaAndTrue[c] \leftarrow 1$;
4      for $s = 1$ to $S$ do
5          $AndNeedStream[c][s] \leftarrow$ FALSE
6      for $i = 1$ to $m_c$ do
7          $LeafCompleted[i] \leftarrow$ TRUE;
8          for $s = 1$ to $S$ do
9              if $n_s < d^{s_s}_{l_{c,i}}$ then
10                 $AndCompleted[c] \leftarrow$ FALSE;
11                 $LeafCompleted[i] \leftarrow$ FALSE;
12                 $AndNeedStream[c][s] \leftarrow$ TRUE;
13         if $LeafCompleted[i]$ then $ProbaAndTrue[c] \leftarrow ProbaAndTrue[c] \times p_{c,i}$
14 $ProbaAllCompletedAndsFalse \leftarrow 1$;
15 for $c = 1$ to $N$ do
16     if $AndCompleted[c]$ then
17         $ProbaAllCompletedAndsFalse \leftarrow$
           $ProbaAllCompletedAndsFalse \times (1 - ProbaAndTrue[c])$
18 DPCOST$[n_1, ..., n_S] \leftarrow +\infty$;
19 NEXTSTREAM$[n_1, ..., n_S] \leftarrow 0$;
20 for $s = 1$ to $S$ do
21     $ProbaAllNeedingAndsFalse \leftarrow 1$;
22     for $c = 1$ to $N$ do
23         if ($\textbf{\textit{not}}$ $AndCompleted[c]$) $\textbf{\textit{and}}$ $AndNeedStream[c][s]$ then
24             $ProbaAllNeedingAndsFalse \leftarrow$
               $ProbaAllNeedingAndsFalse \times (1 - ProbaAndTrue[c])$
25     $ProbaStreamRead \leftarrow$
       $ProbaAllCompletedAndsFalse \times (1 - ProbaAllNeedingAndsFalse)$;
26     $Cost \leftarrow ProbaStreamRead \times c(s_s) +$ DPCOST$[n_1, ..., n_{s-1}, n_s + 1, n_{s+1}, ..., n_S]$;
27     if $Cost <$ DPCOST$[n_1, ..., n_S]$ then
28         DPCOST$[n_1, ..., n_S] \leftarrow Cost$;
29         NEXTSTREAM$[n_1, ..., n_S] \leftarrow s$
```

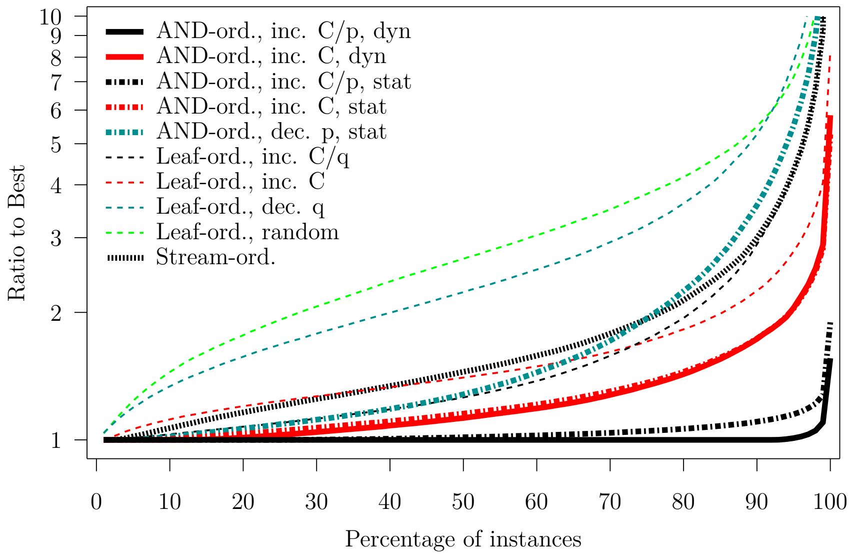


Figure 7.9: Ratio to Best vs. fraction of the instances for which a smaller ratio is achieved, computed over 32,400 random "large" DNF tree instances in the *single-stream* case.

Section 7.4.2 for generating AND tree instances in the *single-stream* case. We generate DNF trees with $N = 2, \ldots, 9$ AND nodes and up to at most 20 leaves and 8 leaves per AND, generating 1,000 random instances for each configuration, for a total of 21,600 instances. For each instance we compute the ratio between the cost achieved by each heuristic and the optimal cost.

Figure 7.8 shows for each heuristic the ratio vs. the fraction of the instances for which the heuristic achieves a lower ratio. For instance, a point at $(80, 2)$ means that the heuristic leads to schedules that are within a factor 2 of optimal for 80% of the instances, and more than a factor 2 away from optimal for 20% of the instances. The better the heuristic the closer its curve is to the horizontal axis. These results include a curve for a heuristic called "Best." This heuristic runs all other heuristics and returns the generated schedule that achieved the lowest expected cost.

The trends in Figure 7.8 are clear. Overall the poorest results are achieved by the leaf-ordered heuristics, with the random such heuristic expectedly being the worst and the increasing $\mathcal{C}$ the best. The AND-ordered heuristics, save for the decreasing p version, lead to the best results overall. For the two AND-ordered heuristics that have both a static and a dynamic version, the dynamic version leads to marginally better results than the static version. Finally, the stream-ordered heuristic leads to poorer results than the best leaf-ordered heuristics, and thus significantly worse results than the best AND-ordered heuristics. Overall, the most effective heuristic is to sort the AND's by increasing $\mathcal{C}/p$.

We also evaluate the heuristics on a set of "large" instances with $N = 2, \ldots, 10$ AND nodes and $m = 5, 10, 15, 20$ leaves per AND node, with 100 random instances per configuration, for a total of 32,400 instances. For most of these instances we cannot tractably compute the optimal cost. Consequently, we compute ratios to the cost achieved by the Best heuristic. Results are shown in Figure 7.9. Essentially, all the observations made on the results for small instances still hold. We conclude that the best approach is to build a depth-first schedule, to sort the AND nodes by the ratio of their costs to probability of success, and to compute these costs dynamically, accounting for previously scheduled AND nodes. This heuristic is the best one in 98.7%, resp. 94.07%, of the cases reported in Figure 7.9, resp. Figure 7.8. It runs in at most 9 seconds on one core of an 2.1 GHz AMD Opteron processor when processing a tree with 10 AND nodes with each 20 leaves.

The source code for all experiments described in this section is available at www.ens-lyon.fr/LIP/ROMA/Data/DataForRR-8373.tgz.

Evaluation results for the *multi-stream* case

As in the previous section, we first evaluate our heuristics on a set of "small" instances for which we can compute optimal schedules using an exponential-time exhaustive search (which is feasible because, due to Theorem 11, it only needs to search over all possible depth-first schedules). Instances are generated using the same method as that described in Section 7.4.3 for generating AND tree instances in the *multi-stream* case. We generate DNF trees with $N = 2, \ldots, 8$ AND nodes and up to at most 16 leaves in total and 7 leaves per AND, generating 100 random instances for each configuration, for a total of 12,600 instances. Results are shown in Figure 7.10, and exhibit clear trends that are similar to those seen in the *single-stream* case. The most effective heuristic is AND-ordered by increasing $\mathcal{C}/p$, dynamic version. This heuristic leads to the optimal solution in 48.98% of the instances, and is the best heuristic in 79.96% of the instances.

These instances are still too large to run the dynamic programming MultiStreamDP heuristic (described in Section 7.5.3) due to its high computational complexity and to its memory requirements. To evaluate this heuristic we generate "very small" instances with $N = 2, \ldots, 5$ AND nodes and up to at most 10 leaves in total and 5 leaves per AND, with a sharing ratio $\rho = 3/2, 2, 3, 4, 5$, or 10 (hence, we only consider the 6 largest of the ratios used in all the other simulations). The number of streams referenced by each leaf is sampled from a uniform distribution over the interval $[1, 3]$ (rather than $[1, 5]$ as in all other *multi-stream* simulations). We generate 100 random instances for each configuration, for a total of 4,800 instances. Results are shown in Figure 7.11. Disappointingly, MultiStreamDP achieves results comparable to the best Leaf-ordered heuristic and poorer than all AND-ordered heuristics except the AND-ordered heuristic with decreasing probability of success (p). We conclude that in spite of being optimal for deciding on a stream order, MultiStreamDP leads to poor results for solving the original problem (and is computationally expensive).

We also evaluate our heuristics on a set of "large" instances with $N = 2, \ldots, 10$ AND nodes and $m = 5, 10, 15, 20$ leaves per AND node, with 100 random instances per configuration, for a total of 32,400 instances. As in the previous section, we compute ratios to the "Best" heuristic since we cannot tractably compute the optimal cost. Results are shown in Figure 7.12. Essentially, all the observations made on the results for small instances hold. The AND-ordered by increasing $\mathcal{C}/p$, dynamic version, heuristic is the best heuristic in 92.16% of the instances.

We conclude that the best approach is to build a depth-first schedule, to sort the AND nodes by the ratio of their costs to probability of success, and to compute these costs dynamically, accounting for previously scheduled AND nodes. This heuristic is the best heuristic in 79.5%, resp. 92.5%, of the cases reported in Figure 7.10, resp. Figure 7.12. On one core of an 2.1 GHz AMD Opteron processor, it runs in at most 8 sec when processing trees with 9 AND nodes with each 15 leaves, and in less than 35 sec when processing tress with 10 AND nodes with each 20 leaves.

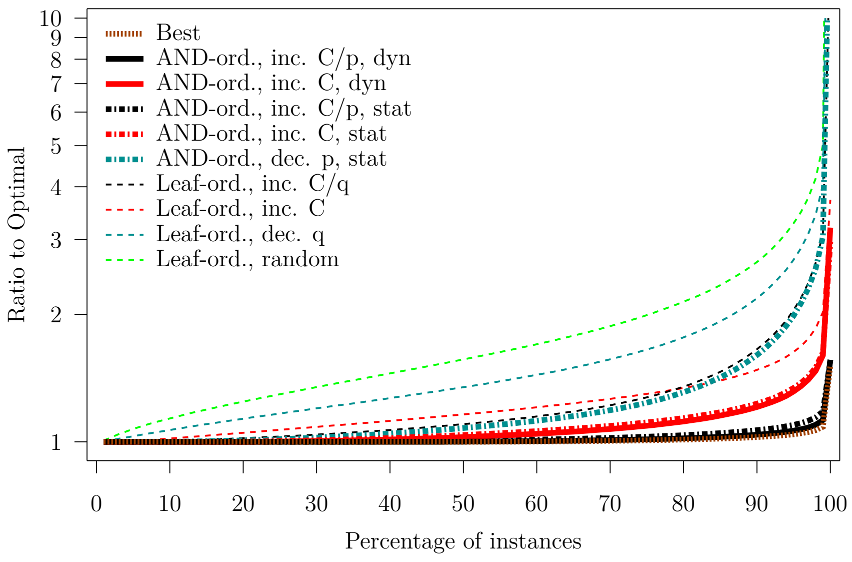


Figure 7.10: Ratio to optimal vs. fraction of the instances for which a smaller ratio is achieved, computed over 16,200 random "small" DNF tree instances in the *multi-stream* case.

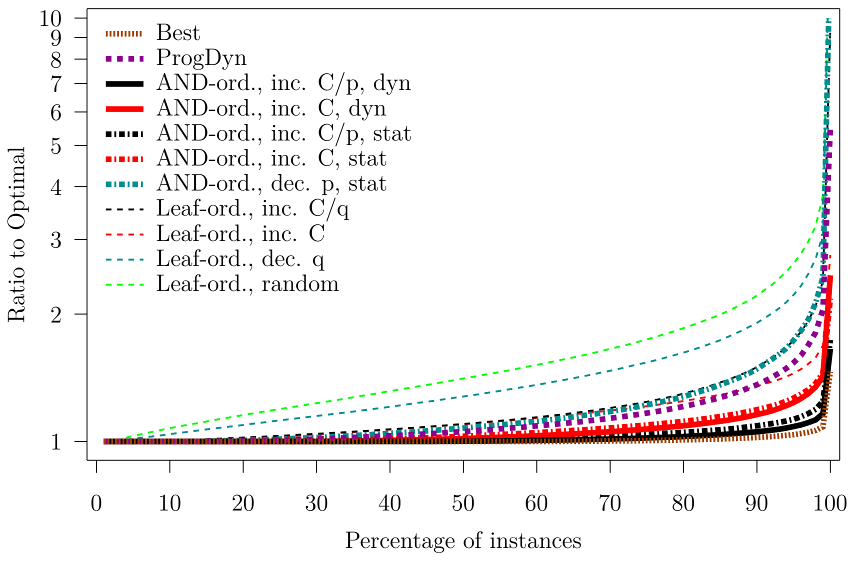


Figure 7.11: Ratio to optimal vs. fraction of the instances for which a smaller ratio is achieved, computed over 4,800 random "very small" DNF tree instances in the *multi-stream* case.

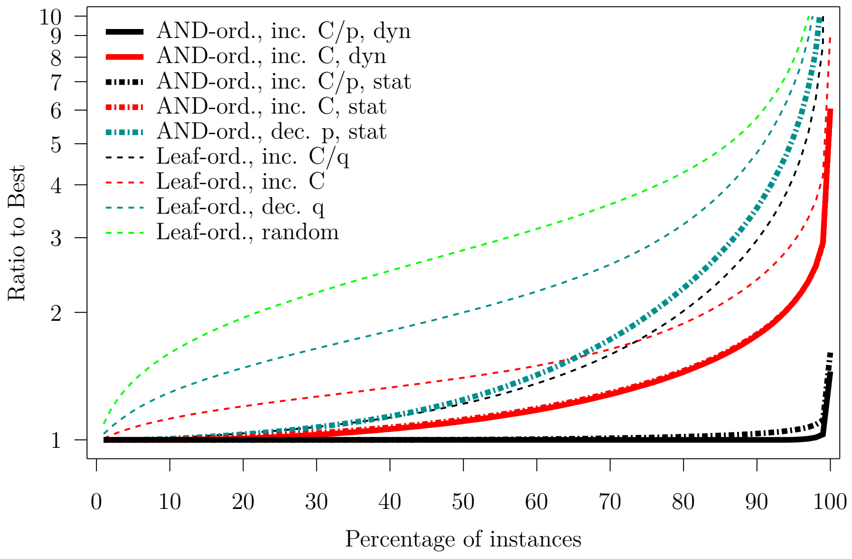


Figure 7.12: Ratio to the Best heuristic vs. fraction of the instances for which a smaller ratio is achieved, computed over 32,400 random "large" DNF tree instances in the *multi-stream* case.

7.6 Conclusion

Motivated by a query processing scenario for sensor data streams, we have studied a version of the Probabilistic And-Or Tree Resolution (PAOTR) problem [105] in which a single leaf may reference multiple data streams and a single data stream may be referenced by multiple leaves. We have given an optimal algorithm in the case of AND trees in the *single-stream* case. We have shown that the problem is NP-complete for AND trees in the *multi-stream* case and for DNF trees in the *single-stream* case. However, we have shown that there is an optimal leaf evaluation order that corresponds to a depth-first traversal. This observation provides inspiration for designing heuristics that produce depth-first traversals. Numerical results obtained for large numbers of random trees show that one of the heuristics we have designed leads to good results in practice.

Conclusion

In this thesis, we have presented two parts dealing with scheduling and optimization problems in probabilistic context. The first part is devoted to the study of resilience on Exascale platforms. In this part we focus on the effecient execution of applications on failure-prone platforms. We have studied the combination of different fault-tolerance protocols with the Coordinated Checkpointing protocol. We typically address questions such as: Given a platform and an application, which fault-tolerance protocols should be used, when, and with which parameters? On the practical side, we conducted simulation experiments to validate each model.

The second part is devoted to the optimization of the expected sensor data acquisition cost when evaluating a query expressed as a tree of boolean operators applied to boolean predicates.

Our main contributions are stated in the following paragraphs.

Summary

Unified Model for Assessing Checkpointing Protocols

In this chapter, we provide a theoretical foundation and a quantitative evaluation of the drawbacks of checkpoint/restart protocols at Exascale. Our model outlines some realistic ranges where hierarchical checkpointing outperforms coordinated checkpointing, thanks to its faster recovery from individual failures. This early result had already been outlined experimentally at smaller scales, but it was difficult to project at future scales. Our results also highlight that significant efforts are required in terms of I/O bandwidth to enable any type of rollback recovery to be competitive and suggests that most research efforts, funding and hardware provisions should be directed to I/O performance rather than improving component reliability.

All results presented in this chapter are corroborated by a set of simulations, we have checked (by an extensive brute-force comparison) that our model could predict near-optimal checkpointing periods for the whole range of the protocol/platform/application combinations; this gives us very good confidence that this model will prove reliable and accurate in other frameworks.

Combining Replication and Coordinated Checkpointing

In this chapter, we have studied group and process replication from a theoretical perspective comparing them to the pure coordinated checkpointing approach in terms of expected application execution times. For group replication, we have proposed a simple yet effective algorithm, we have derived a checkpointing period that minimizes an upper bound on application makespan for exponentially distributed failures and we have proposed a Dynamic Programming approach that computes non-periodic checkpoint dates in a view to minimizing makespan for non-exponentially distributed failures. Finally, we have performed simulation experiments assuming that failures follow Exponential or Weibull distributions and using failure logs from production clusters and the results demonstrate that group replication can be beneficial at large scale.

For process replication, we have derived exact expressions for the Mean Number of Failures To Interruption and the Mean Time To Interruption for arbitrary numbers of replicas assuming Exponential failures, we have extended these results to arbitrary failure distributions, notably obtaining closed-form solutions in the case of Weibull failures. Finally, we have also performed simulation experiments and the results show that the choice of a good checkpointing period is no longer critical when process replication is used.

Combining Fault Prediction and Coordinated Checkpointing

In the first section of this chapter, we assessed the impact of fault prediction on periodic checkpointing using a Predictor with exact prediction dates. We established analytical conditions stating whether a fault prediction should be taken into account or not. More importantly, we proved that the optimal approach is to never trust the predictor in the beginning of a regular period, and to always trust it in the end of the period; the cross-over point $\frac{C_p}{p}$ depends on the time needed to take a proactive checkpoint and on the precision of the predictor. We conducted simulations involving synthetic failure traces following either an Exponential distribution law or a Weibull one. We also used log-based failure traces. Through this extensive experiment setting, we established the accuracy of the model, of its analysis, and of the predicted period (in the presence of a fault predictor). The simulations also show that even a not-so-good fault predictor can lead to quite a significant decrease in the application execution time.

In the second section, we used a Predictor with a prediction window. We proposed a new approach based upon two periodic modes: a regular mode outside prediction windows, and a proactive mode inside prediction windows, whenever the size of these windows is large enough. We were able to fully solve this problem with results corroborated by a full set of simulations.

On the combination of silent error detection and checkpointing

In this chapter, we focused on silent data corruption errors. Contrary to fail-stop failures, such latent errors cannot be detected immediately, and a mechanism to detect them must be provided. We provided a general framework to solve, independently of the verification mechanism, the problem of minimizing the execution time of a schedule. We instantiated the model using realistic scenarios and application/architecture parameters.

Cost-Optimal Execution of Boolean DNF Trees with Shared Streams

In this chapter, we have studied the problem of minimizing the amount of data acquired when evaluating a query expressed as a tree of disjunctive boolean operators applied to boolean predicates of sensor data. We have studied the more general scenario where a stream can occur in multiple leaves of the tree.

We have given an optimal algorithm in the case of AND trees in the *single-stream* case. We have shown that the problem is NP-complete for AND trees in the *multi-stream* case and for DNF trees in the *single-stream* case. However, we have shown that there is an optimal leaf evaluation order that corresponds to a depth-first traversal. This observation provides inspiration for designing heuristics that produce depth-first traversals. Numerical results obtained for large numbers of random trees show that one of the heuristics we have designed leads to good results in practice.

Perspectives

Throughout the thesis, we pointed out at the end of each chapter some future work that remains to be done. In this section, we outline some perspectives of future work that could follow the thesis in the short term. Then we state more general, long-term oriented, research directions.

Resilience on exascale systems

An interesting direction for future work is to investigate the impact of *partial* replication instead of full replication. In this approach, replication would be used only for critical components (e.g., message loggers in uncoordinated checkpoint protocols), while traditional checkpointing would be used for non-critical components. The goal would be to reduce the overhead of replication while still achieving some of its benefits in terms of resilience.

Another direction is to generalize the work on Group replication beyond the case of coordinated checkpointing, for instance to deal with hierarchical checkpointing schemes based on message logging, or with containment domains [28]. Both these techniques alleviate the cost of checkpointing and recovery, and would dramatically decrease checkpointing contention costs.

Cost-Optimal Execution of Boolean Trees with Shared Streams

A possible future direction is to consider so-called *non-linear strategies* [105]. Although in chapter 7 we have considered a schedule as a leaf ordering (called a *linear strategy* in [105]), a more general notion is that of a decision tree in which the next leaf to be evaluated is chosen based on the truth value of the previous evaluated leaf. A practical drawback of a non-linear strategy is that the size of the strategy's description is exponential in the number of tree leaves. In [105], it is shown that in the *read-once* case linear strategies are dominant for DNF trees, meaning that there is always one optimal strategy that is linear. Via a simple counter example it can be shown that this is no longer true in the *shared* see Appendix 7.8), thus motivating the investigation of non-linear strategies.

Another possible future direction is to study the problem for *periodic* query evaluations. In chapter 7 we have considered a single query evaluation, but in practice queries on sensor data streams are evaluated periodically. As a result, data items acquired from previous query evaluations may be re-used for the current evaluation, depending on predicate time-windows and the query evaluation period. The problem is to determine a schedule that minimizes the expected cost of the query evaluation in the long run. The computation of the cost of a given schedule is more complex due to the need to account for data items “left over” from previous query evaluations, and we expect the problem to be more computationally challenging than that studied in this chapter.

General perspectives

Uncoordinated Checkpointing

The Coordinated checkpointing and hierarchical protocols studied in Part 1 suffer a waste in terms of computing resources, whenever living processes have to rollback and recover from a checkpoint in order to tolerate failures. These protocols may also lead to I/O congestion when too many processes are checkpointing at the same time.

An interesting future direction for this thesis would be to study the impact of uncoordinated checkpointing protocol on reliability and identify if this protocol deliver the best performance for a given application/platform pair.

Impact of resilience techniques on energy consumption

An interesting future direction is to study the impact of resilience techniques on energy consumption. Together with fault-tolerance, energy consumption is expected to be a major challenge for exascale machines [92, 95]. A promising next step in this search is the study of the interplay between checkpointing, replication, and energy consumption. By definition, both checkpointing and replication induce additional power consumption, but both techniques lead to faster executions in expectation. There are thus various energy trade-offs to achieve. The key question is to determine the best execution strategy given both an energy budget and a maximum admissible application makespan.

Appendix

7.7 Proof of optimality of SingleStreamGreedy for AND trees

In this section, we present the full proof of Theorem 9.

Proof. We prove the theorem by contradiction. We assume that there exists an instance for which the schedule produced by Algorithm 6, ξ_{greedy}, is not optimal. Among the optimal schedules, let us pick a schedule, ξ_{opt}, which has the longest prefix $\mathbb{P}$ in common with schedule ξ_{greedy}. We consider the first decision (i.e., one recursive call to the algorithm) taken by Algorithm 6 that schedules a leaf that does not belong to $\mathbb{P}$. Let k be the number of leaves scheduled by this decision, and let us denote them $l_{\sigma(1)}$, ..., $l_{\sigma(k)}$, scheduled in this order. Recall each call to the GREEDY algorithm schedules a sequence of leaves that all require data items from the same stream. Furthermore, the scheduled sequence of leaves is a sub-sequence of the ordered sequence of all leaves that require data items from that stream, sorted by increasing number of data items required. Without loss of generality, we assume that $l_{\sigma(1)}$, ..., $l_{\sigma(k)}$ all require items from stream 1. The first of these leaves may belong to $\mathbb{P}$ (as the last leaf occurrences in $\mathbb{P}$). Let $\mathbb{P}'$ be equal to $\mathbb{P}$ minus the leaves $l_{\sigma(1)}$, ..., $l_{\sigma(k)}$. Then, ξ_{greedy} can be written as:

$$\xi_{greedy} = \mathbb{P}', l_{\sigma(1)}, ..., l_{\sigma(k)}, \mathbb{S}. \tag{7.19}$$

In turn, ξ_{opt} can be written $\xi_{opt} = \mathbb{P}', \mathbb{Q}, \mathbb{R}$ where $l_{\sigma(k)}$ is the last leaf of $\mathbb{Q}$. In other words, $\mathbb{Q}$ can be written $L_1 l_{\sigma(1)} L_2 l_{\sigma(2)} ... L_k l_{\sigma(k)}$, where each sequence of leaves L_i, $1 \leq i \leq k$, can be empty. Note that, because of Theorem 15, and because the sequence $l_{\sigma(1)}$, ..., $l_{\sigma(k)}$ is a sub-sequence of the the list of all leaves requiring data items from that stream sorted by increasing number of data items required, none of the L_i sequences can contain a leaf requiring elements from stream 1. Therefore,

$$\xi_{opt} = \mathbb{P}', \mathbb{Q}, \mathbb{R} \text{ where } \mathbb{Q} = L_1 l_{\sigma(1)} L_2 l_{\sigma(2)} ... L_k l_{\sigma(k)} \tag{7.20}$$

From ξ_{greedy} and ξ_{opt}, we build a new schedule, ξ_{new}, defined as

$$\xi_{new} = \mathbb{P}', NewOrder, \mathbb{R} \text{ where } NewOrder = l_{\sigma(1)}, ..., l_{\sigma(k)}, L_1, ..., L_k \tag{7.21}$$

$\mathbb{P}'$, $l_{\sigma(1)}$, ..., $l_{\sigma(k)}$ is a prefix to both ξ_{greedy} and ξ_{new}. This prefix is strictly larger than $\mathbb{P}$ (since $\mathbb{P}$ does not contain $l_{\sigma(k)}$). Therefore, if the cost of ξ_{new} is not greater than that of ξ_{opt}, ξ_{new} is optimal and has a longer prefix in common with ξ_{greedy} than ξ_{new}, which would contradict the definition of ξ_{opt}. We obtain this contradiction by computing the cost of ξ_{new} and showing that it is no larger than that of ξ_{opt}.

Cost notations – To ease the writing of the proof we introduce several notations. If $\mathbb{X}$ is a partial leaf schedule, $P(\mathbb{X})$ denotes the probability that all leaves in $\mathbb{X}$ evaluates to TRUE. In other words, $P(\mathbb{X}) = \prod_{l_i \in \mathbb{X}} p_i$. Let $\mathbb{X}$ and $\mathbb{Y}$ be two disjoint (partial) leaf schedules, i.e., they do not have any leaf in common, such that $\mathbb{X}$ is evaluated right before $\mathbb{Y}$. Then $Cost(\mathbb{Y} \mid \mathbb{X})$ denotes the cost of evaluating $\mathbb{Y}$, assuming that all leaves in $\mathbb{X}$ have evaluated to TRUE. Of course, $Cost(\mathbb{Y} \mid \mathbb{X})$ takes into account all data items acquired during the successful evaluation of $\mathbb{X}$. With these notations, we can now give the costs of ξ_{new} and ξ_{opt} based on their definitions as sequences of partial leaf schedules in Equations (7.20) and (7.21):

$$\begin{aligned}
Cost(\xi_{opt}) &= Cost(\mathbb{P}') + P(\mathbb{P}')Cost(\mathbb{Q} \mid \mathbb{P}') \qquad + P(\mathbb{P}')P(\mathbb{Q})Cost(\mathbb{R} \mid \mathbb{P}', \mathbb{Q}) \\
Cost(\xi_{new}) &= Cost(\mathbb{P}') + P(\mathbb{P}')Cost(NewOrder \mid \mathbb{P}') \\
&\quad + P(\mathbb{P}')P(NewOrder)Cost(\mathbb{R} \mid \mathbb{P}', NewOrder)
\end{aligned}$$

Because $\mathbb{Q}$ and $NewOrder$ contain exactly the same leaves, $P(\mathbb{Q}) = P(NewOrder)$ and $Cost(\mathbb{R} \mid \mathbb{P}', \mathbb{Q}) = Cost(\mathbb{R} \mid \mathbb{P}', NewOrder)$. Therefore,

$$Cost(\xi_{opt}) - Cost(\xi_{new}) = P(\mathbb{P}')\left(Cost(\mathbb{Q} \mid \mathbb{P}') - Cost(NewOrder \mid \mathbb{P}')\right) \tag{7.22}$$

From what precedes, it now suffices to show that $Cost(\mathbb{Q} \mid \mathbb{P}') - Cost(NewOrder \mid \mathbb{P}') \geq 0$ to prove the theorem.

Initial mathematical formulation – We use Proposition 15 to define notations that make it possible to obtain a simple expression for the quantity in Equation 7.22. Consider a stream S and two leaves l_i and l_j that require, respectively, d_{l_i} and d_{l_j} items from stream S, with $d_{l_i} < d_{l_j}$. Then, according to Proposition 15, l_i is always evaluated before l_j in an optimal schedule. The GREEDY algorithm also schedules l_i before l_j. If there does not exist any leaf l_k requiring $d_{l_k} \in [d_{l_i}; d_{l_j}]$ elements from stream s, then each time l_j is evaluated, exactly $d_{l_j} - d_{l_i}$ items are acquired from stream s, because the last d_{l_i} elements of stream s were acquired when l_i was evaluated. In this case, we define a_i as the number of data items that must be acquired when evaluating leaf l_i. Formally,

$$a_i = d_{l_i} - \max\left\{d_{l_j} \mid S(j) = S(i) \text{ and } d_{l_j} < d_{l_i}\right\}$$

Remark: One should note that we can assume without loss of generality that the AND tree does not contain two leaves requiring the exact same number of items from the same stream. If such two leaves exist, then one replaces them by a single leaf with the same data item requirement and with a probability of success that is the product of the probability of success of the two original leaves. This is because once one of the two original leaves has been evaluated then the other one can be evaluated for free.

To ease the writing of the proof, we index the leaves in L_1, ..., L_k according to the stream from which they require data items, and introduce the following additional notations. Let N_i be the number of leaves in $L_1 \cup \ldots \cup L_k$ that require data items from stream i and $l_{i,j}$ be the j-th of these leaves. We then extend the notations defined in Section 7.2 as follows: the probability of success of $l_{i,j}$ is $p_{i,j}$, $l_{i,j}$ requires $d_{l_{i,j}}$ elements from stream $S(i,j)$, etc. $\mu_{(i,j)}$ is the index of the leaf sequence L_p to which leaf $l_{i,j}$ belongs: $l_{i,j} \in L_{\mu_{(i,j)}}$. $Q_{i,j}$ is the product of the success probabilities of the leaves that precede $l_{i,j}$ in $L_{\mu_{(i,j)}}$, Q_m is the product of the success probabilities of all the leaves in L_m, and $\mathcal{Q}_m = \prod_{n=1}^{m} Q_n$. Finally, we define $P_m = \prod_{n=1}^{m} p_{\sigma(n)}$.

With these notations we can now write $Cost(NewOrder \mid \mathbb{P}')$ as:

$$\begin{aligned}
Cost(NewOrder \mid \mathbb{P}') &= \sum_{m=1}^{k}\left(\prod_{n=1}^{m-1} p_{\sigma(n)}\right) a_{\sigma(m)} \\
&+\sum_{i=2}^{S}\sum_{j=1}^{N_i}\left(\prod_{m=1}^{k} p_{\sigma(m)}\right)\left(\prod_{m=1}^{\mu_{(i,j)}-1} Q_m\right) Q_{i,j}a_{i,j}c(S(i,j)) \\
&= \sum_{m=1}^{k} P_{m-1}a_{\sigma(m)} + \sum_{i=2}^{S}\sum_{j=1}^{N_i} P_k \mathcal{Q}_{\mu_{(i,j)}-1} Q_{i,j}a_{i,j}c(S(i,j))\ ,
\end{aligned}$$

and $Cost(\mathbb{Q} \mid \mathbb{P}')$ as:

$$\begin{aligned}
Cost(\mathbb{Q} \mid \mathbb{P}') &= \sum_{m=1}^{k}\left(\prod_{n=1}^{m-1} p_{\sigma(n)}\right)\left(\prod_{n=1}^{m} Q_n\right) a_{\sigma(m)} \\
&+\sum_{i=2}^{S}\sum_{j=1}^{N_i}\left(\prod_{m=1}^{\mu_{(i,j)}-1} p_{\sigma(m)}\right)\left(\prod_{m=1}^{\mu_{(i,j)}-1} Q_m\right) Q_{i,j}a_{i,j}c(S(i,j)) \\
&= \sum_{m=1}^{k} P_{m-1}\mathcal{Q}_m a_{\sigma(m)} + \sum_{i=2}^{S}\sum_{j=1}^{N_i} P_{\mu_{(i,j)}-1} \mathcal{Q}_{\mu_{(i,j)}-1} Q_{i,j}a_{i,j}c(S(i,j))\ .
\end{aligned}$$

Therefore:

$$\begin{aligned}
Cost(\mathbb{Q} \mid \mathbb{P}') - Cost(NewOrder \mid \mathbb{P}') &= \sum_{m=1}^{k} P_{m-1}\mathcal{Q}_m a_{\sigma(m)} + \sum_{i=2}^{S}\sum_{j=1}^{N_i} P_{\mu_{(i,j)}-1} \mathcal{Q}_{\mu_{(i,j)}-1} Q_{i,j}a_{i,j}c(S(i,j)) \\
&- \left(\sum_{m=1}^{k} P_{m-1}a_{\sigma(m)} + \sum_{i=2}^{S}\sum_{j=1}^{N_i} P_k \mathcal{Q}_{\mu_{(i,j)}-1} Q_{i,j}a_{i,j}c(S(i,j))\right) \\
&= \sum_{m=1}^{k} P_{m-1}(\mathcal{Q}_m - 1)a_{\sigma(m)} \\
&+\sum_{i=2}^{S}\sum_{j=1}^{N_i}\left(P_{\mu_{(i,j)}-1} - P_k\right)\mathcal{Q}_{\mu_{(i,j)}-1} Q_{i,j}a_{i,j}c(S(i,j))\ .
\end{aligned}$$

We introduce two additional notations:

$$\alpha_{(i,j)} = \mathcal{Q}_{\mu_{(i,j)}-1}\left(P_{\mu_{(i,j)}-1} - P_k\right) Q_{i,j}\ \text{, and}$$

$$A = \frac{\sum_{m=1}^{k} P_{m-1}a_{\sigma(m)}}{1-P_k}\ ,$$

so that we can finally write the expression for the difference of the two costs:

$$Cost(\mathbb{Q} \mid \mathbb{P}') - Cost(NewOrder \mid \mathbb{P}') = \left(\sum_{m=1}^{k} P_{m-1}\left(\mathcal{Q}_m - 1\right) a_{\sigma(m)}\right) + \left(\sum_{i=2}^{S}\sum_{j=1}^{N_i} \alpha_{(i,j)} a_{i,j}c(S(i,j))\right)\ . \tag{7.23}$$

Accounting for the algorithm's scheduling decisions – The best decision for the GREEDY algorithm was to evaluate at once the leaf sequence $l_{\sigma(1)}, ..., l_{\sigma(k)}$. Therefore, as far as the algorithm is concerned, this was a better decision than evaluating any sequence of leaves from any other stream. More formally, for any stream i, $2 \leq i \leq S$, and the set of the first j leaves of that stream, $1 \leq j \leq N_i$, we have:

$$\frac{\left(\sum_{m=1}^{k}\left(\prod_{n=1}^{m-1} p_{\sigma(n)}\right) a_{\sigma(m)}\right)}{\left(1-\prod_{m=1}^{k} p_{\sigma(m)}\right)}\left(1-\prod_{l=1}^{j} p_{i,l}\right) \leq \left(\sum_{l=1}^{j}\left(\prod_{r=1}^{l-1} p_{i,r}\right) a_{i,l} c(S(i,l))\right) .$$

These equations express the fact that these other sequence of leaves of a *Ratio* value (see Algorithm 6) lower than that of the sequence scheduled by the algorithm, and can be rewritten as:

$$\textsc{Ineq}(i,j): \qquad A\left(1-\prod_{l=1}^{j} p_{i,l}\right) \leq \left(\sum_{l=1}^{j}\left(\prod_{r=1}^{l-1} p_{i,r}\right) a_{i,l} c(S(i,l))\right) . \tag{7.24}$$

Determining multiplying coefficients – To prove the theorem we combine the Inequalities (7.24) obtained for different values of i and j. The idea is to follow a variable elimination process. $\textsc{Ineq}(i, N_i)$ is the only inequality in which a_{i,N_i} appears. We multiply $\textsc{Ineq}(i, N_i)$ by a value λ_{i,N_i} such that, in the resulting inequality, the coefficient of a_{i,N_i} is the same than in Equation (7.23). Next, we multiply $\textsc{Ineq}(i, N_i - 1)$ by a value λ_{i,N_i-1} such that when adding the resulting inequality to the one previously obtained, the coefficient of a_{i,N_i-1} is the same than in Equation (7.23), and so on. This process can be done independently for the different streams as $\textsc{Ineq}(i, j)$ only contains terms relative to stream i.

We define the $\lambda_{i,j}$'s are defined as follows.

$$\lambda_{i,j} = \begin{cases} \dfrac{\alpha_{(i,N_i)}}{\prod_{l=1}^{N_i-1} p_{i,l}} & \text{if } j = N_i \text{ ,} \\ \dfrac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1} p_{i,l}} - \dfrac{\alpha_{(i,j+1)}}{\prod_{l=1}^{j} p_{i,l}} & \text{otherwise.} \end{cases}$$

We will later show that this choice of multipliers enable us to achieve our goal. However, as we want to use the $\lambda_{i,j}$'s as multiplying coefficients for inequalities, we must first show that they are all non-negative. This is evident for the λ_{i,N_i}'s. Let us consider $\lambda_{i,j}$ for $j \in [1; N_i - 1]$:

$$\begin{aligned}
\lambda_{i,j} &= \frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1} p_{i,l}} - \frac{\alpha_{(i,j+1)}}{\prod_{l=1}^{j} p_{i,l}} \\
&= \frac{1}{\prod_{l=1}^{j-1} p_{i,l}} \left(\prod_{m=1}^{\mu_{(i,j)}-1} Q_m \right) \left(\prod_{m=1}^{\mu_{(i,j)}-1} p_{\sigma(m)} \right) \left(1 - \prod_{m=\mu_{(i,j)}}^{k} p_{\sigma(m)} \right) Q_{i,j} \\
&\quad - \frac{1}{\prod_{l=1}^{j} p_{i,l}} \left(\prod_{m=1}^{\mu_{(i,j+1)}-1} Q_m \right) \left(\prod_{m=1}^{\mu_{(i,j+1)}-1} p_{\sigma(m)} \right) \left(1 - \prod_{m=\mu_{(i,j+1)}}^{k} p_{\sigma(m)} \right) Q_{i,j+1} \\
&= \frac{1}{\prod_{l=1}^{j-1} p_{i,l}} \left(\prod_{m=1}^{\mu_{(i,j)}-1} Q_m p_{\sigma(m)} \right) \\
&\qquad \times \left[\left(1 - \prod_{m=\mu_{(i,j)}}^{k} p_{\sigma(m)} \right) Q_{i,j} - \left(\prod_{m=\mu_{(i,j)}}^{\mu_{(i,j+1)}-1} Q_m p_{\sigma(m)} \right) \left(1 - \prod_{m=\mu_{(i,j+1)}}^{k} p_{\sigma(m)} \right) \frac{Q_{i,j+1}}{p_{i,j}} \right]
\end{aligned}$$

Let us first consider the case $\mu_{(i,j+1)} = \mu_{(i,j)}$. Then the above equation can be rewritten:

$$\lambda_{i,j} = \frac{1}{\prod_{l=1}^{j-1} p_{i,l}} \left(\prod_{m=1}^{\mu_{(i,j)}-1} Q_m p_{\sigma(m)} \right) \left(1 - \prod_{m=\mu_{(i,j)}}^{k} p_{\sigma(m)} \right) \left[Q_{i,j} - \frac{Q_{i,j+1}}{p_{i,j}} \right]$$

By definition, $Q_{i,j+1}$ is the product of the probabilities of success of all the leaves that are evaluated before the leaf $l_{i,j+1}$ is evaluated. By definition of the numbering of the leaves, this includes at least all the leaves that are evaluated before leaf $l_{i,j}$ is evaluated and leaf $l_{i,j}$. As all probabilities are less than or equal to 1, this implies that $Q_{i,j+1} \leq Q_{i,j} p_{i,j}$, and therefore that $\lambda_{i,j} \geq 0$.

We now consider the other case: $\mu_{(i,j+1)} > \mu_{(i,j)}$. Then, $Q_{\mu_{(i,j)}}$ is of the form $Q_{i,j} p_{i,j} X$ where X is the product of the probabilities of success of the leaves appearing in $L_{\mu_{(i,j)}}$ after the leaf $l_{i,j}$. Therefore, $Q_{i,j} p_{i,j} \geq Q_{\mu_{(i,j)}}$. As for all $i \in [1; S]$, $0 \leq p_{\sigma(i)} \leq 1$, $\prod_{m=\mu_{(i,j+1)}}^{k} p_{\sigma(m)} \geq \prod_{m=\mu_{(i,j)}}^{k} p_{\sigma(m)}$ and $1 - \prod_{m=\mu_{(i,j+1)}}^{k} p_{\sigma(m)} \leq 1 - \prod_{m=\mu_{(i,j)}}^{k} p_{\sigma(m)}$, $p_{\sigma(\mu_{(i,j)})} (\prod_{m=\mu_{(i,j)}+1}^{\mu_{(i,j+1)}-1} Q_m p_{\sigma(m)}) Q_{i,j+1} \leq 1$ because it is a product of probabilities. Using these inequalities, we have:

$$\begin{aligned}
\left(1 - \prod_{m=\mu_{(i,j)}}^{k} p_{\sigma(m)}\right) Q_{i,j} - \left(\prod_{m=\mu_{(i,j)}}^{\mu_{(i,j+1)}-1} Q_m p_{\sigma(m)}\right) \left(1 - \prod_{m=\mu_{(i,j+1)}}^{k} p_{\sigma(m)}\right) \frac{Q_{i,j+1}}{p_{i,j}} &\geq \\
\left(1 - \prod_{m=\mu_{(i,j)}}^{k} p_{\sigma(m)}\right) Q_{i,j} - \left(\prod_{m=\mu_{(i,j)}}^{\mu_{(i,j+1)}-1} Q_m p_{\sigma(m)}\right) \left(1 - \prod_{m=\mu_{(i,j)}}^{k} p_{\sigma(m)}\right) \frac{Q_{i,j+1}}{p_{i,j}} &= \\
\left(1 - \prod_{m=\mu_{(i,j)}}^{k} p_{\sigma(m)}\right) \left(Q_{i,j} - \left(\prod_{m=\mu_{(i,j)}}^{\mu_{(i,j+1)}-1} Q_m p_{\sigma(m)}\right) \frac{Q_{i,j+1}}{p_{i,j}}\right) &\geq \\
\left(1 - \prod_{m=\mu_{(i,j)}}^{k} p_{\sigma(m)}\right) \left(Q_{i,j} - Q_{\mu_{(i,j)}} \left(p_{\sigma(\mu_{(i,j)})} (\prod_{m=\mu_{(i,j)}+1}^{\mu_{(i,j+1)}-1} Q_m p_{\sigma(m)}) Q_{i,j+1}\right) \frac{1}{p_{i,j}}\right) &\geq \\
\left(1 - \prod_{m=\mu_{(i,j)}}^{k} p_{\sigma(m)}\right) \left(Q_{i,j} - Q_{\mu_{(i,j)}} \frac{1}{p_{i,j}}\right) &\geq 0
\end{aligned}$$

Therefore, all the $\lambda_{i,j}$'s are non-negative.

Combining the inequalities – For a given couple of values (i, j), with $2 \leq i \leq S$ and $1 \leq j \leq N_i$, let $\textsc{Ineq}(i, j)$ be Inequality (7.24) defined for (i, j). Because all the $\lambda_{i,j}$'s are non-negative, we can form the inequality:

$$\sum_{i=2}^{S} \sum_{j=1}^{N_i} (\lambda_{i,j} \times \textsc{Ineq}(i, j)) \tag{7.25}$$

We now show that Inequality (7.25) leads to:

$$Cost(\mathbb{Q} \mid \mathbb{P}') - Cost(NewOrder \mid \mathbb{P}') \geq \sum_{m=1}^{k} P_{m-1} \left(\mathcal{Q}_m - 1\right) a_{\sigma(m)} + A \sum_{i=2}^{S} \left(\sum_{j=1}^{N_i} \alpha_{(i,j)} \left(1 - p_{i,j}\right) \right) \tag{7.26}$$

To prove the Inequality (7.26), we consider the terms relative to stream i in Inequality (7.25):

$$\begin{aligned} & \sum_{j=1}^{N_i} (\lambda_{i,j} \times \textsc{Ineq}(i, j)) \quad \Leftrightarrow \\ A \sum_{j=1}^{N_i} \lambda_{i,j} \left(1 - \prod_{l=1}^{j} p_{i,l}\right) \quad & \leq \quad \sum_{j=1}^{N_i} \lambda_{i,j} \left(\sum_{l=1}^{j} \left(\prod_{r=1}^{l-1} p_{i,r} \right) a_{i,l} c(S(i, l)) \right) \end{aligned} \tag{7.27}$$

We start by considering the left-hand side of this inequality.

$$\begin{aligned}
&\sum_{j=1}^{N_i} \lambda_{i,j} \left(1 - \prod_{l=1}^{j} p_{i,l}\right) &=\\
&\left(\sum_{j=1}^{N_i-1} \lambda_{i,j} \left(1 - \prod_{l=1}^{j} p_{i,l}\right)\right) + \lambda_{i,N_i} \left(1 - \prod_{l=1}^{N_i} p_{i,l}\right) &=\\
&\left(\sum_{j=1}^{N_i-1} \left(\frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1} p_{i,l}} - \frac{\alpha_{(i,j+1)}}{\prod_{l=1}^{j} p_{i,l}}\right) \left(1 - \prod_{l=1}^{j} p_{i,l}\right)\right) + \frac{\alpha_{(i,N_i)}}{\prod_{l=1}^{N_i-1} p_{i,l}} \left(1 - \prod_{l=1}^{N_i} p_{i,l}\right) &=\\
&\left(\sum_{j=1}^{N_i-1} \frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1} p_{i,l}} \left(1 - \prod_{l=1}^{j} p_{i,l}\right)\right) - \left(\sum_{j=1}^{N_i-1} \frac{\alpha_{(i,j+1)}}{\prod_{l=1}^{j} p_{i,l}} \left(1 - \prod_{l=1}^{j} p_{i,l}\right)\right) + \frac{\alpha_{(i,N_i)}}{\prod_{l=1}^{N_i-1} p_{i,l}} \left(1 - \prod_{l=1}^{N_i} p_{i,l}\right) &=\\
&\left(\sum_{j=1}^{N_i-1} \frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1} p_{i,l}} \left(1 - \prod_{l=1}^{j} p_{i,l}\right)\right) - \left(\sum_{j=2}^{N_i} \frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1} p_{i,l}} \left(1 - \prod_{l=1}^{j-1} p_{i,l}\right)\right) + \frac{\alpha_{(i,N_i)}}{\prod_{l=1}^{N_i-1} p_{i,l}} \left(1 - \prod_{l=1}^{N_i} p_{i,l}\right) &=\\
&\left(\sum_{j=1}^{N_i} \frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1} p_{i,l}} \left(1 - \prod_{l=1}^{j} p_{i,l}\right)\right) - \left(\sum_{j=2}^{N_i} \frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1} p_{i,l}} \left(1 - \prod_{l=1}^{j-1} p_{i,l}\right)\right) &=\\
&\alpha_{(i,1)} \left(1 - p_{i,1}\right) + \left(\sum_{j=2}^{N_i} \frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1} p_{i,l}} \left(1 - \prod_{l=1}^{j} p_{i,l}\right)\right) - \left(\sum_{j=2}^{N_i} \frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1} p_{i,l}} \left(1 - \prod_{l=1}^{j-1} p_{i,l}\right)\right) &=\\
&\alpha_{(i,1)} \left(1 - p_{i,1}\right) + \sum_{j=2}^{N_i} \frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1} p_{i,l}} \left(\prod_{l=1}^{j-1} p_{i,l} - \prod_{l=1}^{j} p_{i,l}\right) &=\\
&\alpha_{(i,1)} \left(1 - p_{i,1}\right) + \sum_{j=2}^{N_i} \alpha_{(i,j)} \left(1 - p_{i,j}\right) &=\\
&\sum_{j=1}^{N_i} \alpha_{(i,j)} \left(1 - p_{i,j}\right)\ .
\end{aligned}$$

Therefore,

$$A \sum_{j=1}^{N_i} \lambda_{i,j} \left(1 - \prod_{l=1}^{j} p_{i,l}\right) = A \left(\sum_{j=1}^{N_i} \alpha_{(i,j)} \left(1 - p_{i,j}\right)\right)\ . \tag{7.28}$$

We now focus on the right-hand side of the inequality:

$$
\begin{aligned}
&\sum_{j=1}^{N_i}\lambda_{i,j}\left(\sum_{l=1}^{j}\left(\prod_{r=1}^{l-1}p_{i,r}\right)a_{i,l}c(S(i,l))\right)\\
&=\left(\sum_{j=1}^{N_i-1}\lambda_{i,j}\left(\sum_{l=1}^{j}\left(\prod_{r=1}^{l-1}p_{i,r}\right)a_{i,l}c(S(i,l))\right)\right)+\lambda_{i,N_i}\left(\sum_{l=1}^{N_i}\left(\prod_{r=1}^{l-1}p_{i,r}\right)a_{i,l}c(S(i,l))\right)\\
&=\left(\sum_{j=1}^{N_i-1}\left(\frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1}p_{i,l}}-\frac{\alpha_{(i,j+1)}}{\prod_{l=1}^{j}p_{i,l}}\right)\left(\sum_{l=1}^{j}\left(\prod_{r=1}^{l-1}p_{i,r}\right)a_{i,l}c(S(i,l))\right)\right)+\frac{\alpha_{(i,N_i)}}{\prod_{l=1}^{N_i-1}p_{i,l}}\left(\sum_{l=1}^{N_i}\left(\prod_{r=1}^{l-1}p_{i,r}\right)a_{i,l}c(S(i,l))\right)\\
&=\left(\sum_{j=1}^{N_i-1}\frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1}p_{i,l}}\left(\sum_{l=1}^{j}\left(\prod_{r=1}^{l-1}p_{i,r}\right)a_{i,l}c(S(i,l))\right)\right)-\left(\sum_{j=1}^{N_i-1}\frac{\alpha_{(i,j+1)}}{\prod_{l=1}^{j}p_{i,l}}\left(\sum_{l=1}^{j}\left(\prod_{r=1}^{l-1}p_{i,r}\right)a_{i,l}c(S(i,l))\right)\right)\\
&\qquad+\frac{\alpha_{(i,N_i)}}{\prod_{l=1}^{N_i-1}p_{i,l}}\left(\sum_{l=1}^{N_i}\left(\prod_{r=1}^{l-1}p_{i,r}\right)a_{i,l}c(S(i,l))\right)\\
&=\left(\sum_{j=1}^{N_i}\frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1}p_{i,l}}\left(\sum_{l=1}^{j}\left(\prod_{r=1}^{l-1}p_{i,r}\right)a_{i,l}c(S(i,l))\right)\right)-\left(\sum_{j=1}^{N_i-1}\frac{\alpha_{(i,j+1)}}{\prod_{l=1}^{j}p_{i,l}}\left(\sum_{l=1}^{j}\left(\prod_{r=1}^{l-1}p_{i,r}\right)a_{i,l}c(S(i,l))\right)\right)\\
&\left(\sum_{j=1}^{N_i}\frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1}p_{i,l}}\left(\sum_{l=1}^{j}\left(\prod_{r=1}^{l-1}p_{i,r}\right)a_{i,l}c(S(i,l))\right)\right)-\left(\sum_{j=2}^{N_i}\frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1}p_{i,l}}\left(\sum_{l=1}^{j-1}\left(\prod_{r=1}^{l-1}p_{i,r}\right)a_{i,l}c(S(i,l))\right)\right)\\
&=\alpha_{(i,1)}a_{i,1}c(S(i,1))+\left(\sum_{j=2}^{N_i}\frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1}p_{i,l}}\left(\sum_{l=1}^{j}\left(\prod_{r=1}^{l-1}p_{i,r}\right)a_{i,l}c(S(i,l))\right)\right)\\
&\qquad-\left(\sum_{j=2}^{N_i}\frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1}p_{i,l}}\left(\sum_{l=1}^{j-1}\left(\prod_{r=1}^{l-1}p_{i,r}\right)a_{i,l}c(S(i,l))\right)\right)\\
&=\alpha_{(i,1)}a_{i,1}c(S(i,1))+\sum_{j=2}^{N_i}\frac{\alpha_{(i,j)}}{\prod_{l=1}^{j-1}p_{i,l}}\left(\prod_{r=1}^{j-1}p_{i,r}\right)a_{i,j}c(S(i,j))\\
&=\sum_{j=1}^{N_i}\alpha_{(i,j)}a_{i,j}c(S(i,j))\ .
\end{aligned}
$$

Therefore

$$\sum_{j=1}^{N_i}\lambda_{i,j}\left(\sum_{l=1}^{j}\left(\prod_{r=1}^{l-1}p_{i,r}\right)a_{i,l}c(S(i,l))\right)=\left(\sum_{j=1}^{N_i}\alpha_{(i,j)}a_{i,j}c(S(i,j))\right)\ . \tag{7.29}$$

By combining Inequality (7.27) with Equations (7.28) and (7.29), and by summing over all streams, we obtain:

$$A\sum_{i=2}^{S}\left(\sum_{j=1}^{N_i}\alpha_{(i,j)}\left(1-p_{i,j}\right)\right)\leq\sum_{i=2}^{S}\left(\sum_{j=1}^{N_i}\alpha_{(i,j)}a_{i,j}c(S(i,j))\right)\ . \tag{7.30}$$

Using Equation (7.23) and Inequality (7.30), we obtain:

$$\mathit{Cost}(\mathbb{Q}\mid\mathbb{P}')-\mathit{Cost}(\mathit{NewOrder}\mid\mathbb{P}')\geq\sum_{m=1}^{k}P_{m-1}\left(\mathcal{Q}_m-1\right)a_{\sigma(m)}+A\sum_{i=2}^{S}\left(\sum_{j=1}^{N_i}\alpha_{(i,j)}\left(1-p_{i,j}\right)\right)\ . \tag{7.31}$$

Completing the proof – We want to prove that the right-hand side of Inequality (7.31) is non-negative, i.e., that the following inequality holds:

$$\sum_{m=1}^{k} P_{m-1}\left(\mathcal{Q}_m - 1\right) a_{\sigma(m)} + A \sum_{i=2}^{S} \left(\sum_{j=1}^{N_i} \alpha_{(i,j)} \left(1 - p_{i,j}\right) \right) \geq 0 \,. \tag{7.32}$$

Because of Inequality (7.31), this will enable us to conclude. Let

$$A_n = \sum_{m=1}^{n} P_{m-1} a_{\sigma(m)} \,.$$

Therefore, $A = \frac{A_k}{(1-P_k)}$. We start by focusing on the first term of Inequality (7.32). We prove that:

$$\sum_{m=1}^{k} P_{m-1}\left(\mathcal{Q}_m - 1\right) a_{\sigma(m)} \geq A \left(\left(\sum_{i=1}^{k} (\mathcal{Q}_i - \mathcal{Q}_{i-1}) P_{i-1} \right) + P_k (1 - \mathcal{Q}_k) \right) \,. \tag{7.33}$$

$$\begin{aligned}
&\sum_{m=1}^{k} P_{m-1}\left(\mathcal{Q}_m - 1\right) a_{\sigma(m)} \\
&= \sum_{m=1}^{k} \left(\mathcal{Q}_m - 1\right) \left(\sum_{n=1}^{m} P_{n-1} a_n - \sum_{n=1}^{m-1} P_{n-1} a_n \right) \\
&= \sum_{m=1}^{k} (\mathcal{Q}_m - 1)(A_m - A_{m-1}) \\
&= \left(\sum_{m=1}^{k} \mathcal{Q}_m (A_m - A_{m-1}) \right) - \left(\sum_{m=1}^{k} (A_m - A_{m-1}) \right) \\
&= \left(\sum_{m=1}^{k} (\mathcal{Q}_m A_m - \mathcal{Q}_m A_{m-1}) \right) - (A_k - A_0) \\
&= \left(\sum_{m=1}^{k} (\mathcal{Q}_m A_m - \mathcal{Q}_{m-1} A_{m-1} + \mathcal{Q}_{m-1} A_{m-1} - \mathcal{Q}_m A_{m-1}) \right) - A_k \\
&= \left(\sum_{m=1}^{k} (\mathcal{Q}_m A_m - \mathcal{Q}_{m-1} A_{m-1}) \right) + \left(\sum_{m=1}^{k} (\mathcal{Q}_{m-1} A_{m-1} - \mathcal{Q}_m A_{m-1}) \right) - A_k \\
&= (\mathcal{Q}_k A_k - \mathcal{Q}_0 A_0) + \left(\sum_{m=1}^{k} (\mathcal{Q}_{m-1} - \mathcal{Q}_m) A_{m-1} \right) - A_k \\
&= \left(\sum_{m=1}^{k} (\mathcal{Q}_{m-1} - \mathcal{Q}_m) A_{m-1} \right) + (\mathcal{Q}_k - 1) A_k = \left(\sum_{m=1}^{k} (\mathcal{Q}_{m-1} - \mathcal{Q}_m) A_{m-1} \right) + (\mathcal{Q}_k - 1) A (1 - P_k) \,.
\end{aligned}$$

By hypothesis, the best decision for the greedy algorithm was to read at once the leaves a_1, ..., a_k. Therefore, this was better than reading any other sequence of leaves from stream 1. So, for any value of $m \leq k$,

$$A\left(1 - P_{m-1}\right) \leq A_{m-1} \,.$$

Because $\mathcal{Q}_m = \mathcal{Q}_{m-1} Q_m$ with $Q_m \in [0;1]$, then $\mathcal{Q}_{m-1} - \mathcal{Q}_m \geq 0$. Therefore, we have:

$$\begin{aligned}
&\sum_{m=1}^{k} (\mathcal{Q}_{m-1} - \mathcal{Q}_m) A_{m-1} + A(1-P_k)(\mathcal{Q}_k - 1) \\
&\geq A\left[\left(\sum_{m=1}^{k} (\mathcal{Q}_{m-1} - \mathcal{Q}_m)(1-P_{m-1})\right) + (1-P_k)(\mathcal{Q}_k - 1)\right] \\
&= A\left[\left(\sum_{m=1}^{k} (\mathcal{Q}_{m-1} - \mathcal{Q}_m)\right) - \left(\sum_{m=1}^{k} (\mathcal{Q}_{m-1} - \mathcal{Q}_m) P_{m-1}\right) + (1-P_k)(\mathcal{Q}_k - 1)\right] \\
&= A\left[(\mathcal{Q}_0 - \mathcal{Q}_k) + \left(\sum_{m=1}^{k} (\mathcal{Q}_m - \mathcal{Q}_{m-1}) P_{m-1}\right) + (1-P_k)(\mathcal{Q}_k - 1)\right] \\
&= A\left[1 - \mathcal{Q}_k + \left(\sum_{m=1}^{k} (\mathcal{Q}_m - \mathcal{Q}_{m-1}) P_{m-1}\right) + (1-P_k)(\mathcal{Q}_k - 1)\right] \\
&= A\left[\left(\sum_{m=1}^{k} (\mathcal{Q}_m - \mathcal{Q}_{m-1}) P_{m-1}\right) - P_k(\mathcal{Q}_k - 1)\right] .
\end{aligned}$$

We now focus on the second term of Inequality (7.32). We prove must prove that:

$$\sum_{i=2}^{S} \sum_{j=1}^{N_i} \alpha_{(i,j)} (1 - p_{i,j}) = \left(\sum_{m=1}^{k} \mathcal{Q}_{m-1} P_{m-1} (1 - Q_m)\right) - P_k (1 - \mathcal{Q}_k) . \tag{7.34}$$

We have:

$$\begin{aligned}
&\sum_{i=2}^{S} \sum_{j=1}^{N_i} \alpha_{(i,j)} (1 - p_{i,j}) \\
&= \sum_{i=2}^{S} \sum_{j=1}^{N_i} \left(\prod_{m=1}^{\mu_{(i,j)}-1} Q_m p_{\sigma(m)}\right) \left(1 - \prod_{m=\mu_{(i,j)}}^{k} p_{\sigma(m)}\right) Q_{i,j} (1 - p_{i,j}) \\
&= \sum_{i=2}^{S} \sum_{j=1}^{N_i} \mathcal{Q}_{\mu_{(i,j)}-1} P_{\mu_{(i,j)}-1} \left(1 - \prod_{m=\mu_{(i,j)}}^{k} p_{\sigma(m)}\right) Q_{i,j} (1 - p_{i,j}) \\
&= \left(\sum_{i=2}^{S} \sum_{j=1}^{N_i} \mathcal{Q}_{\mu_{(i,j)}-1} P_{\mu_{(i,j)}-1} Q_{i,j} (1 - p_{i,j})\right) - \left(\sum_{i=2}^{S} \sum_{j=1}^{N_i} \mathcal{Q}_{\mu_{(i,j)}-1} P_{\mu_{(i,j)}-1} \left(\prod_{m=\mu_{(i,j)}}^{k} p_{\sigma(m)}\right) Q_{i,j} (1 - p_{i,j})\right) \\
&= \left(\sum_{i=2}^{S} \sum_{j=1}^{N_i} \mathcal{Q}_{\mu_{(i,j)}-1} P_{\mu_{(i,j)}-1} Q_{i,j} (1 - p_{i,j})\right) - \left(\sum_{i=2}^{S} \sum_{j=1}^{N_i} \mathcal{Q}_{\mu_{(i,j)}-1} P_k Q_{i,j} (1 - p_{i,j})\right) \\
&= \left(\sum_{i=2}^{S} \sum_{j=1}^{N_i} \mathcal{Q}_{\mu_{(i,j)}-1} P_{\mu_{(i,j)}-1} Q_{i,j} (1 - p_{i,j})\right) - \left(P_k \sum_{i=2}^{S} \sum_{j=1}^{N_i} \mathcal{Q}_{\mu_{(i,j)}-1} Q_{i,j} (1 - p_{i,j})\right) .
\end{aligned}$$

We concentrate on the second term and its meaning. For any stream i and any of its leaves j, $\mathcal{Q}_{\mu_{(i,j)}-1} Q_{i,j}$ is the probability of success of all the leaves evaluated before the studied leaf (not considering the leaves of other streams), and $\mathcal{Q}_{\mu_{(i,j)}-1}Q_{i,j}p_{i,j}$ is the same probability right after the evaluation of the studied leaf. Therefore, in the inner sum all terms cancel out except the first one, that is the probability of success if no leaf had been evaluated so far, and the last one, that is the probability of success if all the leaves have been evaluated. Formally, we have:

$$\sum_{i=2}^{S}\sum_{j=1}^{N_i}\mathcal{Q}_{\mu_{(i,j)}-1}Q_{i,j}(1-p_{i,j}) = 1-\left(\prod_{m=1}^{k}Q_m\right) = 1-\mathcal{Q}_k\ . \tag{7.35}$$

Therefore,

$$\begin{aligned}
&\sum_{i=2}^{S}\sum_{j=1}^{N_i}\alpha_{(i,j)}(1-p_{i,j})\\
&=\left(\sum_{i=2}^{S}\sum_{j=1}^{N_i}\mathcal{Q}_{\mu_{(i,j)}-1}P_{\mu_{(i,j)}-1}Q_{i,j}(1-p_{i,j})\right)-\left(P_k\sum_{i=2}^{S}\sum_{j=1}^{N_i}\mathcal{Q}_{\mu_{(i,j)}-1}Q_{i,j}(1-p_{i,j})\right)\\
&=\left(\sum_{i=2}^{S}\sum_{j=1}^{N_i}\mathcal{Q}_{\mu_{(i,j)}-1}P_{\mu_{(i,j)}-1}Q_{i,j}(1-p_{i,j})\right)-P_k(1-\mathcal{Q}_k)\\
&=\left(\sum_{i=2}^{S}\sum_{j=1}^{N_i}(\mathcal{Q}_{\mu_{(i,j)}-1}P_{\mu_{(i,j)}-1}Q_{i,j}-\mathcal{Q}_{\mu_{(i,j)}-1}P_{\mu_{(i,j)}-1}Q_{i,j}p_{i,j})\right)-P_k(1-\mathcal{Q}_k)\\
&=\left(\sum_{m=1}^{k}\sum_{\substack{(i,j)\text{ s.t.}\\ \mu_{(i,j)}=m}}(\mathcal{Q}_{m-1}P_{m-1}Q_{i,j}-\mathcal{Q}_{m-1}P_{m-1}Q_{i,j}p_{i,j})\right)-P_k(1-\mathcal{Q}_k)\\
&=\left(\sum_{m=1}^{k}\mathcal{Q}_{m-1}P_{m-1}\sum_{\substack{(i,j)\text{ s.t.}\\ \mu_{(i,j)}=m}}(Q_{i,j}-Q_{i,j}p_{i,j})\right)-P_k(1-\mathcal{Q}_k)\\
&=\left(\sum_{m=1}^{k}\mathcal{Q}_{m-1}P_{m-1}(1-Q_m)\right)-P_k(1-\mathcal{Q}_k)\ .
\end{aligned}$$

The last equality above is established using the same type of reasoning as the one we used to establish Equation (7.35).

We now combine Inequality (7.33) with Equation (7.34):

$$\begin{aligned}
&\sum_{m=1}^{k} (\mathcal{Q}_m - 1)\, P_{m-1} a_{\sigma(m)} + A \sum_{i=2}^{S} \sum_{j=1}^{N_i} \alpha_{(i,j)}(1 - p_{i,j}) \\
&\geq A\left(\left(\sum_{m=1}^{k} (\mathcal{Q}_m - \mathcal{Q}_{m-1}) P_{m-1}\right) + P_k(1 - \mathcal{Q}_k)\right) + A\left(\left(\sum_{m=1}^{k} \mathcal{Q}_{m-1} P_{m-1}(1 - Q_m)\right) - P_k(1 - \mathcal{Q}_k)\right) \\
&= A\left(\left(\sum_{m=1}^{k} (\mathcal{Q}_m - \mathcal{Q}_{m-1}) P_{m-1}\right) + P_k(1 - \mathcal{Q}_k) + \left(\sum_{m=1}^{k} \mathcal{Q}_{m-1} P_{m-1}(1 - Q_m)\right) - P_k(1 - \mathcal{Q}_k)\right) \\
&= A\left(\left(\sum_{m=1}^{k} (\mathcal{Q}_m - \mathcal{Q}_{m-1}) P_{m-1}\right) + \left(\sum_{m=1}^{k} \mathcal{Q}_{m-1} P_{m-1}(1 - Q_m)\right)\right) \\
&= A\left(\sum_{m=1}^{k} (\mathcal{Q}_m P_{m-1} - \mathcal{Q}_{m-1} P_{m-1} + \mathcal{Q}_{m-1} P_{m-1} - \mathcal{Q}_{m-1} P_{m-1} Q_m)\right) \\
&= 0\ ,
\end{aligned}$$

because $\mathcal{Q}_{m-1} Q_m = \mathcal{Q}_m$. We have thus established Inequality (7.32), which concludes the proof. ■

7.8 Dominant Linear Strategy Counter-Example

A more general notion than a schedule, called a *strategy*, is described in [105]. Although it may seem counter-intuitive, the processing of a query does not have to follow a defined ordering of the leaves. Instead, a strategy is a decision tree in which the next leaf to be evaluated is chosen based on the truth value of the leaves that have been evaluated previously. A *schedule*, as defined in this work, is a particular kind of strategy, termed a "linear strategy" in [105]. Therein, the authors prove that for some problem instances the best linear strategy can be far from being the optimal strategy. Although interesting from a theoretical standpoint, a practical drawback of a non-linear strategy is that the size of its description is exponential in the number of tree leaves. Instead, a linear strategy, or schedule, is simply an ordering of the leaves, with a description size linear in the number of tree leaves. This severe drawback explains why we have not considered non-linear strategies in this work.

However, from a theoretical point of view, it is interesting to ask the following question: while linear strategies are dominant (among all possible strategies) for DNF trees in the *read-once* case [105], is it still the case in the *shared* case? We show that the answer is negative by building a counter-example.

Consider three streams, A, B, and C, with per data item costs $c(A) = 1$, $c(B) = 1.1$, and $c(C) = 1$. Consider the query tree in Figure 7.13, where for each leaf is indicated the success probability, the stream needed, and the number of data items required from that stream. We first compute the best schedule (i.e., leaf ordering). The cost of schedule l_1, l_2, l_3, l_4 is

$$\begin{aligned}
c(A) + p_1(c(B) + (1 - p_2)c(B)) + (1 - p_1)(2c(B)) \\
+ (1 - p_1 p_2)(1 - p_3)(2c(C)) = 1.95 < 2
\end{aligned}$$

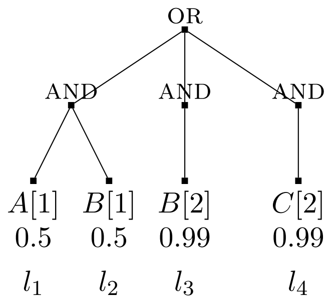


Figure 7.13: Example DNF tree for which the best schedule has larger cost than a non-linear strategy.

The cost of any schedule starting with l_3 or l_4 is at least 2. The cost of schedule l_1, l_2, l_4, l_3 is larger than

$$c(A) + p_1(c(B) + (1 - p_1p_2)(2c(C)) = 2.15 > 2.$$

Finally, if we start with the first AND node, it is always better to start with leaf l_1 whose cost is 0. Altogether, the best schedule is l_1, l_2, l_3, l_4, of cost 1.95.

Now, consider the non-linear strategy that evaluates l_1 first and then:
— if l_1 evaluates to TRUE, proceeds with l_2, l_3, and l_4, just as in the optimal schedule;
— if l_1 evaluates to FALSE, proceeds with l_4, l_3, and l_2.
The cost of this strategy is

$$\begin{aligned} &c(A) + \\ p_1[(c(B) + (1-p_2)c(B)) &+ (1-p_2)(1-p_3)(2c(C))] \\ +(1-p_1)[2c(C)) &+ (1-p_4)(2c(B))] = 1.851, \end{aligned}$$

which is lower than that of the best schedule.

Determining the optimal non-linear strategy for a DNF tree in the *shared* model is an open problem. Unless some structural property of this strategy can be proven, the space requires to describe this optimal non-linear strategy is unknown (and likely exponential).

Bibliography

[1] "Top500 list - June 2014." http://www.top500.org/lists/2014/06/.

[2] G. Zheng, X. Ni, and L. V. Kale, "A scalable double in-memory checkpoint and restart scheme towards exascale," in *Dependable Systems and Networks Workshops (DSN-W)*, 2012.

[3] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Fault tolerant preconditioned conjugate gradient for sparse linear system solution," in *Proc. 26th Int. Conf. on Supercomputing*, ICS '12, ACM, 2012.

[4] A. R. Benson, S. Schmit, and R. Schreiber, "Silent error detection in numerical time-stepping schemes.," *CoRR*, vol. abs/1312.2674, 2013.

[5] P. Sao and R. Vuduc, "Self-stabilizing iterative solvers," in *Proc. Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '13, ACM, 2013.

[6] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proceedings of the ACM/IEEE SC Conference*, pp. 1–11, 2010.

[7] X. Ouyang, S. Marcarelli, and D. K. Panda, "Enhancing checkpoint performance with staging io and ssd," in *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os*, SNAPI '10, (Washington, DC, USA), pp. 13–20, IEEE Computer Society, 2010.

[8] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, (New York, NY, USA), pp. 277–286, ACM, 2004.

[9] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, (Washington, DC, USA), pp. 9–, IEEE Computer Society, 2005.

[10] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, pp. 972–986, 1998.

[11] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka, "Distributed diskless checkpoint for large scale systems," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, (Washington, DC, USA), pp. 63–72, IEEE Computer Society, 2010.

[12] S. Sumimoto, "An Overview of Fujitsu's Lustre Based File System." Lustre Filesystem Users' Group Meeting, Orlando, USA., April 2011.

[13] L. E. Cannon, *A cellular computer to implement the Kalman filter algorithm.* PhD thesis, Montana State University, 1969.

[14] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien, "Checkpointing strategies for parallel jobs," in *Proc. ACM/IEEE Conf. on Supercomputing*, 2011.

[15] M. Wu, X.-H. Sun, and H. Jin, "Performance under failures of high-end computing," in *Proc. ACM/IEEE Conf. on Supercomputing*, 2007.

[16] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello, "MPICH-V: a multiprotocol fault tolerant MPI," *IJHPCA*, vol. 20, no. 3, pp. 319–333, 2006.

[17] S. Rao, L. Alvisi, H. M. Viny, and D. C. Sciences, "Egida: An extensible toolkit for low-overhead fault-tolerance," in *In Symposium on Fault-Tolerant Computing*, pp. 48–55, Press, 1999.

[18] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Survey*, vol. 34, pp. 375–408, 2002.

[19] S. Rao, L. Alvisi, and H. M. Vin, "The cost of recovery in message logging protocols," in *17th Symposium on Reliable Distributed Systems (SRDS)*, pp. 10–18, IEEE CS Press, October 1998.

[20] A. Bouteiller, T. Herault, G. Bosilca, and J. J. Dongarra, "Correlated set coordination in fault tolerant message logging protocols," in *Proc. of Euro-Par'11 (II)*, vol. 6853 of *LNCS*, pp. 51–64, Springer, 2011.

[21] C. L. M. Esteban Meneses and L. V. Kalé, "Team-based message logging: Preliminary results," in *Workshop Resilience in Clusters, Clouds, and Grids (CCGRID 2010).*, 2010.

[22] A. Guermouche, T. Ropars, M. Snir, and F. Cappello, "HydEE: Failure Containment without Event Logging for Large Scale Send-Deterministic MPI Applications," in *IPDPS'12*, IEEE, 2012.

[23] A. Bouteiller, G. Bosilca, and J. Dongarra, "Redesigning the message logging model for high performance," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 16, pp. 2196–2211, 2010.

[24] E. Elnozahy and J. Plank, "Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 97—108, 2004.

[25] 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 *Proc. of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pp. 30—46, 2007.

[26] B. Schroeder and G. A. Gibson, "Understanding Failures in Petascale Computers," *Journal of Physics: Conference Series*, vol. 78, no. 1, 2007.

[27] R. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, and F. Cappello, "Modeling and Tolerating Heterogeneous Failures on Large Parallel System," in *Proc. of the IEEE/ACM Supercomputing Conference (SC)*, 2011.

[28] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez, "Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems," in *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis SC'12*, ACM Press, 2012.

[29] X.-J. Yang, Z. Wang, J. Xue, and Y. Zhou, "The Reliability Wall for Exascale Supercomputing," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 767–779, 2012.

[30] W. Jones, J. Daly, and N. DeBardeleben, “Impact of sub-optimal checkpoint intervals on application efficiency in computational clusters,” in *HPDC’10*, pp. 276–279, ACM, 2010.

[31] S. Yi, D. Kondo, B. Kim, G. Park, and Y. Cho, “Using Replication and Checkpointing for Reliable Task Management in Computational Grids,” in *Proc. of the International Conference on High Performance Computing & Simulation*, 2010.

[32] C. Engelmann, H. H. Ong, and S. L. Scorr, “The case for modular redundancy in large-scale highh performance computing systems,” in *Proc. of the 8th IASTED Infernational Conference on Parallel and Distributed Computing and Networks (PDCN)*, pp. 189–194, 2009.

[33] Z. Zheng and Z. Lan, “Reliability-aware scalability models for high performance computing,” in *Proc. of the IEEE Conference on Cluster Computing*, 2009.

[34] C. Engelmann and B. Swen, “Redundant execution of HPC applications with MR-MPI,” in *PDCN*, IASTED, 2011.

[35] 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*, pp. 124–133, Springer-Verlag, 2009.

[36] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, “Combining partial redundancy and checkpointing for HPC,” in *ICDCS’12*, IEEE, 2012.

[37] 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 (a DSN workshop)*, IEEE, 2012.

[38] C. George and S. S. Vadhiyar, “Adft: An adaptive framework for fault tolerance on large scale systems using application malleability,” *Procedia Computer Science*, vol. 9, pp. 166 – 175, 2012.

[39] L. S. Blackford, J. Choi, A. Cleary, E. D’Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users’ Guide*. SIAM, 1997.

[40] M. Pinedo, *Scheduling: theory, algorithms, and systems (3rd edition)*. Springer, 2008.

[41] K. Venkatesh, “Analysis of Dependencies of Checkpoint Cost and Checkpoint Interval of Fault Tolerant MPI Applications,” *Analysis*, vol. 2, no. 08, pp. 2690–2697, 2010.

[42] F. Gärtner, “Fundamentals of fault-tolerant distributed computing in asynchronous environments,” *ACM Computing Surveys*, vol. 31, no. 1, 1999.

[43] G. Amdahl, “The validity of the single processor approach to achieving large scale computing capabilities,” in *AFIPS Conf. Proceedings*, vol. 30, pp. 483–485, AFIPS Press, 1967.

[44] P. Flajolet, P. J. Grabner, P. Kirschenhofer, and H. Prodinger, “On Ramanujan’s Q-Function,” *J. Computational and Applied Mathematics*, vol. 58, pp. 103–116, 1995.

[45] R. Riesen, K. Ferreira, and J. Stearley, “See applications run and throughput jump: The case for redundant computing in HPC,” in *Proc. of the Dependable Systems and Networks Workshops*, pp. 29–34, 2010.

[46] G. Zheng, X. Ni, and L. Kale, “A scalable double in-memory checkpoint and restart scheme towards exascale,” in *Dependable Systems and Networks Workshops (DSN-W)*, 2012.

[47] E. W. Fulp, G. A. Fink, and J. N. Haack, "Predicting computer system failures using support vector machines," in *Proceedings of the First USENIX conference on Analysis of system logs*, USENIX Association, 2008.

[48] A. Gainaru, F. Cappello, and W. Kramer, "Taming of the shrew: Modeling the normal and faulty behavior of large-scale hpc systems," in *Proc. IPDPS'12*, 2012.

[49] A. Gainaru, F. Cappello, W. Kramer, and M. Snir, "Fault prediction under the microscope - a closer look into hpc systems," in *Proceedings of the ACM/IEEE conference on SuperComputing (SC)*, 2012.

[50] Y. Liang, Y. Zhang, H. Xiong, and R. K. Sahoo, "Failure prediction in ibm bluegene/l event logs," in *ICDM*, pp. 583–588, 2007.

[51] L. Yu, Z. Zheng, Z. Lan, and S. Coghlan, "Practical online failure prediction for blue gene/p: Period-based vs event-driven," in *Dependable Systems and Networks Workshops (DSN-W)*, pp. 259–264, 2011.

[52] Z. Zheng, Z. Lan, R. Gupta, S. Coghlan, and P. Beckman, "A practical failure prediction with location and lead time for blue gene/p," in *Dependable Systems and Networks Workshops (DSN-W)*, pp. 15–22, 2010.

[53] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Comm. of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[54] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *FGCS*, vol. 22, no. 3, pp. 303–312, 2004.

[55] N. Kolettis and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *FTCS '95*, (Washington, DC, USA), p. 381, IEEE CS, 1995.

[56] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert, "Proactive management of software aging," *IBM J. Res. Dev.*, vol. 45, no. 2, pp. 311–332, 2001.

[57] J. Hong, S. Kim, Y. Cho, H. Yeom, and T. Park, "On the choice of checkpoint interval using memory usage profile and adaptive time series analysis," in *Proc. Pacific Rim Int. Symp. on Dependable Computing*, IEEE Computer Society, 2001.

[58] S. M. Ross, *Introduction to Probability Models, Tenth Edition.* Academic Press, 2009.

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

[60] J. Wingstrom, *Overcoming The Difficulties Created By The Volatile Nature Of Desktop Grids Through Understanding, Prediction And Redundancy.* PhD thesis, University of Hawai'i at Manoa, 2009.

[61] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien, "Checkpointing strategies for parallel jobs," in *Proceedings of SC'11*, 2011.

[62] Y. Robert, F. Vivien, and D. Zaidouni, "On the complexity of scheduling checkpoints for computational workflows," in *FTXS'2012, the Workshop on Fault-Tolerance for HPC at Extreme Scale, in conjunction with the 42nd Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN 2012)*, IEEE Computer Society Press, 2012.

[63] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *Proc. of DSN*, pp. 249–258, 2006.

[64] T. Heath, R. P. Martin, and T. D. Nguyen, "Improving cluster availability using workstation validation," *SIGMETRICS Perf. Eval. Rev.*, vol. 30, no. 1, 2002.

[65] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. Scott, "An optimal checkpoint/restart model for a large scale high performance computing system," in *IPDPS'08*, IEEE, 2008.

[66] 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.

[67] D. Kondo, B. Javadi, A. Iosup, and D. Epema, "The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems," *Cluster Computing and the Grid, IEEE International Symposium on*, vol. 0, pp. 398–407, 2010.

[68] F. Cappello, H. Casanova, and Y. Robert, "Preventive migration vs. preventive checkpointing for extreme scale supercomputers," *Parallel Processing Letters*, vol. 21, no. 2, pp. 111–132, 2011.

[69] 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 the 2011 ACM/IEEE Conf. on Supercomputing*, 2011.

[70] Y. Li, Z. Lan, P. Gujrati, and X. Sun, "Fault-aware runtime strategies for high-performance computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 20, no. 4, pp. 460–473, 2009.

[71] M. Bouguerra, A. Gainaru, L. Gomez, F. Cappello, S. Matsuoka, and N. Maruyama, "Improving the computing efficiency of HPC systems using a combination of proactive and preventive checkpointing," in *Proceedings of the 27th International Parallel & Distributed Processing Symposium (IPDPS'13)*, pp. 501–512, IEEE, May 2013.

[72] "Raw data and simulator source code." http://graal.ens-lyon.fr/~fvivien/DATA/predictionwindow/.

[73] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM J. Res. Dev.*, vol. 6, no. 2, pp. 200–209, 1962.

[74] G. Lu, Z. Zheng, and A. A. Chien, "When is multi-version checkpointing needed," in *3rd Workshop for Fault-tolerance at Extreme Scale (FTXS)*, ACM Press, 2013. https://sites.google.com/site/uchicagolssg/lssg/research/gvr.

[75] Y. Ling, J. Mi, and X. Lin, "A variational calculus approach to optimal checkpoint placement," *IEEE Trans. on computers*, pp. 699–708, 2001.

[76] T. Ozaki, T. Dohi, H. Okamura, and N. Kaio, "Distribution-free checkpoint placement algorithms based on min-max principle," *IEEE TDSC*, pp. 130–140, 2006.

[77] M.-S. Bouguerra, T. Gautier, D. Trystram, and J.-M. Vincent, "A flexible checkpoint/restart model in distributed systems," in *PPAM*, vol. 6067 of *LNCS*, pp. 206–215, 2010.

[78] S. Toueg and O. Babaoglu, "On the optimum checkpoint selection problem," *SIAM J. Computing*, vol. 13, no. 3, pp. 630–649, 1984.

[79] M.-S. Bouguerra, D. Trystram, and F. Wagner, "Complexity Analysis of Checkpoint Scheduling with Variable Costs," *IEEE Transactions on Computers*, vol. 99, no. PrePrints, 2012.

[80] J. S. Plank and M. G. Thomason, "Processor allocation and checkpoint interval selection in cluster computing systems," *J. of Parallel and Distributed Computing*, vol. 61, p. 1590, 2001.

[81] H. Jin, Y. Chen, H. Zhu, and X.-H. Sun, "Optimizing HPC Fault-Tolerant Environment: An Analytical Approach," in *Parallel Processing (ICPP), 2010*, pp. 525–534, 2010.

[82] L. Wang, P. Karthik, Z. Kalbarczyk, R. Iyer, L. Votta, C. Vick, and A. Wood, "Modeling Coordinated Checkpointing for Large-Scale Supercomputers," in *Proc. of ICDSN*, pp. 812–821, 2005.

[83] R. Oldfield, S. Arunagiri, P. Teller, S. Seelam, M. Varela, R. Riesen, and P. Roth, "Modeling the impact of checkpoints on next-generation systems," in *Proc. of IEEE MSST*, pp. 30–46, 2007.

[84] Z. Zheng and Z. Lan, "Reliability-aware scalability models for high performance computing," in *Proc. of IEEE Cluster*, 2009.

[85] G. Bronevetsky and B. de Supinski, "Soft error vulnerability of iterative linear algebra methods," in *Proc. 22nd Int. Conf. on Supercomputing*, ICS '08, pp. 155–164, ACM, 2008.

[86] M. Heroux and M. Hoemmen, "Fault-tolerant iterative methods via selective reliability," Research report SAND2011-3915 C, Sandia National Laboratories, 2011.

[87] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based fault tolerance applied to high performance computing," *J. Parallel and Distributed Computing*, vol. 69, no. 4, pp. 410 –416, 2009.

[88] 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," *SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 111–122, 2012.

[89] 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. of the ACM/IEEE SC Int. Conf.*, 2012.

[90] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. 33, no. 6, pp. 518–528, 1984.

[91] F. Cappello, A. Geist, B. Gropp, L. V. Kalé, B. Kramer, and M. Snir, "Toward Exascale Resilience," *Int. Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

[92] J. Dongarra, P. Beckman, P. Aerts, F. Cappello, T. Lippert, S. Matsuoka, P. Messina, T. Moore, R. Stevens, A. Trefethen, and M. Valero, "The international exascale software project: a call to cooperative action by the global high-performance community," *Int. Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 309–322, 2009.

[93] "Maple sheets for the experiments." `http://graal.ens-lyon.fr/~yrobert/error-detection/`.

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

[95] V. Sarkar *et al.*, "Exascale software study: Software challenges in extreme scale systems," 2009. White paper available at: `http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECSS%20report%20101909.pdf`.

[96] K. M. Chandy and L. Lamport, “Distributed snapshots: Determining global states of distributed systems,” in *Transactions on Computer Systems*, vol. 3(1), pp. 63–75, ACM, February 1985.

[97] T. Ünlüyurt, “Sequential testing of complex systems: a review,” *Discrete Applied Mathematics*, vol. 142, no. 1-3, pp. 189–205, 2004.

[98] Y. Jiang, H. Qiu, M. McCartney, W. Halfond, F. Bai, D. Grimm, and R. Govindan, “Flexible and Efficient Sensor Fusion for Automotive Apps,” Technical Report 13-939, Univ. of Southern California, 2013. `http://www.cs.usc.edu/assets/007/89156.pdf`.

[99] “The SHIMMER sensor platform.” `http://shimmer-research.com`, 2013.

[100] L. Lim, A. Misra, and T. Mo, “Adaptive Data Acquisition Strategies for Energy-Efficient Smartphone-based Continuous Processing of Sensor Streams,” *Distributed Parallel Databases*, vol. 31, no. 2, pp. 321–351, 2013.

[101] E. Miluzzo, “Sensing Meets Mobile Social Networks: The Design, Implementation and Evaluation of the CenceMe Application,” in *Proc. of ACM Conf. on Embedded Networked Sensor Systems*, 2008.

[102] I. Mohomed, A. Misra, M. Ebling, and W. Jerome, “Context-Aware and Personalized Event Filtering for Low-Overhead Continuous Remote Health Monitoring,” in *Proc. of the IEEE Intl. Symp. on a World of Wireless Mobile and Multimedia Networks*, 2008.

[103] S. Gaonkar, J. Li, R. Roy Choudhury, L. Cox, and A. Schmidt, “Micro-Blog: Sharing and Querying Content through Mobile Phones and Social Participation,” in *Proc. of the ACM Intl. Conf. on Mobile Systems, Applications, and Services*, 2008.

[104] D. E. Smith, “Controlling backward inference,” *Artificial Intelligence*, vol. 39, no. 2, pp. 145—208, 1989.

[105] R. Greiner, R. Hayward, M. Jankowska, and M. Molloy, “Finding Optimal Satisficing Strategies for And-Or Trees,” *Artificial Intelligence*, vol. 170, no. 1, pp. 19–58, 2006.

[106] F. Cicalese, E. Laber, and A. Medeiros Saettler, “Decision Trees for the efficient evaluation of discrete functions: worst case and expected case analysis,” *ArXiv e-prints*, September 2013.

[107] D. Golovin, A. Krause, and D. Ray, “Near-optimal bayesian active learning with noisy observations,” in *Advances in Neural Information Processing Systems 23* (J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, eds.), pp. 766–774, 2010.

[108] G. Bellala, S. Bhavnani, and C. Scott, “Group-based active query selection for rapid diagnosis in time-critical situations,” *Information Theory, IEEE Transactions on*, vol. 58, no. 1, pp. 459–478, 2012.

[109] M. Charikar, R. Fagin, V. Guruswami, J. Kleinberg, P. Raghavan, and A. Sahai, “Query strategies for priced information,” *J. Comput. Syst. Sci.*, vol. 64, pp. 785–819, June 2002.

[110] F. Cicalese and E. S. Laber, “On the competitive ratio of evaluating priced functions,” *J. ACM*, vol. 58, pp. 9:1–9:40, June 2011.

[111] H. Kaplan, E. Kushilevitzi, and Y. Mansour, “Learning with attribute costs,” *The 37th ACM Symposium on Theory of Computing STOC’05*, 2005.

[112] B. M. E. Moret, “Decision trees and diagrams,” *ACM Computing Surveys*, vol. 14, 1982.

Publications

Articles in international refereed journals

[J1] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni, “Checkpointing algorithms and fault prediction,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 2, pp. 2048–2064, 2014.

[J2] G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni, “Unified model for assessing checkpointing protocols at extreme- scale,” *Concurrency and Computation: Practice and Experience*, 2013.

[J3] M. Bougeret, H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni, “Using group replication for resilience on exascale systems,” *International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 208–222, May 2014.

Articles in international refereed conferences

[C1] H. Casanova, L. Lim, Y. Robert, F. Vivien, and D. Zaidouni, “Cost-optimal execution of boolean query trees with shared streams,” in *IPDPS 2014-28th IEEE International Parallel and Distributed Processing Symposium*, 2014.

[C2] G. Aupy, A. Benoit, T. Hérault, Y. Robert, F. Vivien, and D. Zaidouni, “On the combination of silent error detection and checkpointing,” in *PRDC 2013-19th Pacific Rim International Symposium on Dependable Computing*, 2013.

[C3] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni, “Checkpointing strategies with prediction windows,” in *PRDC 2013-19th Pacific Rim International Symposium on Dependable Computing*, 2013.

[C4] Y. Robert, F. Vivien, and D. Zaidouni, “On the complexity of scheduling checkpoints for computational workflows,” in *FTXS 2012-2nd Workshop on Fault-Tolerance for HPC at Extreme Scale*, 2012.

Book chapter

[B1] H. Casanova, F. Vivien, and D. Zaidouni, *Fault-tolerant techniques for High-Performance Computing*, ch. Using replication for resilience on exascale systems. Springer, 2015. To appear.

Research reports

[RR1] H. Casanova, L. Lim, Y. Robert, F. Vivien, and D. Zaidouni, “Cost-Optimal Execution of Boolean DNF Trees with Shared Streams,” Tech. Rep. RR-8616, INRIA, Nov. 2014.

[RR2] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni, “Comments on ”Improving the computing efficiency of HPC systems using a combination of proactive and preventive checkpoint”,” Rapport de recherche RR-8318, INRIA, June 2013.

[RR3] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni, “Checkpointing strategies with prediction windows,” Tech. Rep. RR-8239, INRIA, Feb. 2013.

[RR4] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni, “Checkpointing algorithms and fault prediction,” Tech. Rep. RR-8237, INRIA, Feb. 2013.

[RR5] G. Bosilca, A. Bouteiller, E. Brunet, C. Franck, J. Dongarra, G. Amina, T. Hérault, Y. Robert, F. Vivien, and D. Zaidouni, “Unified Model for Assessing Checkpointing Protocols at Extreme-Scale,” Tech. Rep. RR-7950, INRIA, Oct. 2012.

[RR6] H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni, “Combining Process Replication and Checkpointing for Resilience on Exascale Systems,” Tech. Rep. RR-7951, INRIA, 2012.

[RR7] M. Bougeret, H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni, “Using group replication for resilience on exascale systems,” Tech. Rep. RR-7876, INRIA, 2012.

[RR8] Y. Robert, F. Vivien, and D. Zaidouni, “On the complexity of scheduling checkpoints for computational workflows,” Tech. Rep. RR-7907, INRIA, 2012.