LUND UNIVERSITY

Fault Tolerance for Real-Time Systems: Analysis and Optimization of Roll-back Recovery with Checkpointing

Nikolov, Dimitar

2014

Fault Tolerance for Real-Time Systems

Analysis and Optimization of Roll-back Recovery with Checkpointing

Dimitar Nikolov

LUND UNIVERSITY

Doctoral Dissertation
Fault-tolerant computing
Lund, January 2015

Dimitar Nikolov
Department of Electrical and Information Technology
Fault-tolerant computing
Lund University
P.O. Box 118, 221 00 Lund, Sweden

Series of licentiate and doctoral dissertations
ISSN 1654-790X; No. 68
ISBN 978-91-7623-146-3 (print)
ISBN 978-91-7623-147-0 (pdf)

Typeset in Palatino and Helvetica using LaTeX 2_ε.
Printed in Sweden by Tryckeriet i E-huset, Lund University, Lund.

To my family

"Everything is possible. The impossible just takes longer."

Dan Brown
novelist

Popular Science Summary

In the modern world of today, we are surrounded with many electronic devices which offer previously unseen performance. These devices constitute a large part of the everyday consumer electronics such as laptops, tablets, smart phones, etc., but are also used in wide variety of domains such as automotive industry, avionics, medicine, etc. The constant demand for high performance has resulted in a rapid development of semiconductor technologies. Technology scaling has pushed the boundaries enabling fabrication of miniature devices. With such miniature devices, it is possible to integrate an entire system on a single chip, commonly referred to as System-on-Chip. For example, in a recent smart phone, the size of such chip is less than one square centimeter, and within this area, the number of transistors (the fundamental building block of modern electronic devices) is over one billion. This example shows that the gains of technology scaling are enormous. However, this comes at a cost, and that is that devices manufactured in the latest technologies may be affected by errors which cause malfunction. Lately, soft errors have been named as one of the most serious threats to computer systems designed in the latest semiconductor technologies.

Soft errors occur as a result of a particular type of faults, known as transient faults. Transient faults have a limited lifetime, namely these faults occur, remain present for a short time, but disappear afterwards. However, these faults, despite their short duration, often result in soft errors that may lead to a system failure. Therefore, soft errors have a significant impact on the reliability of the computer systems manufactured in the latest semiconductor technologies. While in the past soft errors were only a threat for devices which operate in harsh environments such as nuclear plants where high level of radiation exists, or avionics where at higher altitudes the cosmic radiation is higher, nowadays soft errors are a threat for all devices irrespective of the

operational environment. The reason for this is that the major source of soft errors is the radiation of alpha-particles which are emitted from the package of the device itself. In the digital world where everything revolves around bits (a binary digit "1" or "0"), a transient fault can be interpreted as the outside force that flips a bit from "1" to "0", or vice-versa, and the flipped bit is the representation of a soft error. If a soft error occurs in your smart phone, tablet or laptop you easily handle it by restarting the device. However, what happens if such an error occurs in a vital component of an airplane or a nuclear reactor? Can we simply restart?

The field of research which tries to give answers to the previous questions is called **Fault Tolerance**. As the name suggests, fault tolerance enables correct operation of a device even in the presence of faults (errors). As a research topic, fault tolerance has been established along with the rise of the very first devices used in safety-critical applications such as avionics. Lately, the popularity of fault tolerance has been increased, especially when the manufacturing process has moved down to deep sub-micron semiconductor technologies where the size of the transistors has shrunk substantially, and their operation has become more susceptible to soft errors. To enable correct operation in the presence of errors, fault tolerance provides techniques that are capable of error-detection, *i.e.* detect the presence of errors, and error-recovery, *i.e.* recover the system from errors. Usually, this is achieved by introducing a hardware and time redundancy.

Hardware redundancy techniques cope with errors by designing devices or computer systems, such that multiple copies (replicas) of the physical building blocks are used. Every block performs a given operation and provides some kind of an output based on some inputs. If two identical copies process the same inputs, it is expected that they would both produce the same outputs. While these techniques ensure correct operation in the presence of errors, the main drawback is that these techniques are rather expensive. The cost of a device which contains multiple replicas is higher.

In contrast to the expensive hardware redundancy techniques, time redundancy techniques cope with errors by repeating the same operation utilizing the given hardware resources. The correct output is obtained by repeating the same operation at least twice. This increases the time required to obtain the final outcome, and therefore it results in much higher time overhead.

Roll-back Recovery with Checkpointing (RRC) is a well-known fault tolerance technique that efficiently copes with soft errors. Unlike traditional time redundancy techniques, where upon error detection the program is restarted from the beginning, RRC stores checkpoints (intermediate states of the execution of the program), and when errors are detected, it forces the program to roll-back to the latest stored checkpoint. The advantage of this technique over other fault tolerance techniques is that it does not require a substantial

amount of hardware redundancy. However, the major drawback of RRC is that it introduces time overhead that depends on the number of checkpoints that are used. Thus, RRC introduces a time overhead that may have a negative impact on the computer system where it is used.

In general, computer systems are classified into real-time systems (RTSs) and non-RTSs, depending on the requirement to meet time constraints. For RTSs, the correct operation is defined as producing the correct output while satisfying a given time constraint (deadline). Depending on the consequences when deadlines are violated, RTSs are divided into soft and hard RTS. For soft RTSs, the consequences are not very severe. One example of a soft RTS can be a mobile phone where eventual deadline violation results in a dropped call. On the other hand, violating the deadlines in hard RTSs usually results in catastrophic consequences. An example of a hard RTS can be the braking control in a vehicle. RTSs are also affected by soft errors, and therefore there is a need to employ fault tolerance in RTSs as well. However, special consideration should be taken when employing fault tolerance in RTSs, due to the fact that fault tolerance usually introduces a time overhead.

The time overhead due to usage of fault tolerance in RTSs, may result in a missed deadline. To mitigate this effect, it is important to optimize the usage of fault tolerance in RTSs. The optimization objectives differ among soft and hard RTSs. For soft RTSs, where eventual deadline violation results in some performance degradation, it is more important to minimize the average execution time (the average time needed for the operation to complete), while for hard RTSs, where it is crucial to meet the deadlines, it is more important to maximize the probability that the deadlines are met.

During the early design stage of an RTS, a designer of an RTS receives a specification of the RTS that is to be implemented. During this stage, the designer needs to explore different fault tolerance techniques and choose the one that satisfies the given specification requirements. To assist the designer in the decision making process, in this thesis, we provide an optimization framework for RRC when used in RTSs. By using this framework, the designer of an RTS can first decide if RRC is a suitable fault tolerance technique for the RTS that is to be implemented, and then if RRC is applicable, the designer can acquire knowledge on the number of checkpoints that need to be used and how these checkpoints need to be distributed. The proposed optimization framework considers multiple optimization objectives that are important for RTSs. In particular, for soft RTSs the optimization framework considers optimization of RRC with respect to AET. For hard RTSs, the optimization framework considers optimization of RRC with the goal to maximize the Level of Confidence (LoC), *i.e.* the probability that the deadlines are met. Since a specification of an RTS that is to be implemented may include some reliability requirements, in this thesis, we have introduced the concept

of Guaranteed Completion Time, *i.e.* a completion time that satisfies a given reliability (LoC) constraint. The Guaranteed Completion Time varies with the number of checkpoints used in RRC. Therefore, the optimization framework considers optimization of RRC with respect to Guaranteed Completion Time.

Abstract

Increasing soft error rates in recent semiconductor technologies enforce the usage of fault tolerance. While fault tolerance enables correct operation in the presence of soft errors, it usually introduces a time overhead. The time overhead is particularly important for a group of computer systems referred to as real-time systems (RTSs) where correct operation is defined as producing the correct result of a computation while satisfying given time constraints (deadlines). Depending on the consequences when the deadlines are violated, RTSs are classified into soft and hard RTSs. While violating deadlines in soft RTSs usually results in some performance degradation, violating deadlines in hard RTSs results in catastrophic consequences. To determine if deadlines are met, RTSs are analyzed with respect to average execution time (AET) and worst case execution time (WCET), where AET is used for soft RTSs, and WCET is used for hard RTSs. When fault tolerance is employed in both soft and hard RTSs, the time overhead caused due to usage of fault tolerance may be the reason that deadlines in RTSs are violated. Therefore, there is a need to optimize the usage of fault tolerance in RTSs.

To enable correct operation of RTSs in the presence of soft errors, in this thesis we consider a fault tolerance technique, Roll-back Recovery with Checkpointing (RRC), that efficiently copes with soft errors. The major drawback of RRC is that it introduces a time overhead which depends on the number of checkpoints that are used in RRC. Depending on how the checkpoints are distributed throughout the execution of the job, we consider the two checkpointing schemes: equidistant checkpointing, where the checkpoints are evenly distributed, and non-equidistant checkpointing, where the checkpoints are not evenly distributed. The goal of this thesis is to provide an optimization framework for RRC when used in RTSs while considering different optimization objectives which are important for RTSs.

The purpose of such an optimization framework is to assist the designer of an RTS during the early design stage, when the designer needs to explore different fault tolerance techniques, and choose a particular fault tolerance technique that meets the specification requirements for the RTS that is to be implemented. By using the optimization framework presented in this thesis, the designer of an RTS can acquire knowledge if RRC is a suitable fault tolerance technique for the RTS which needs to be implemented. The proposed optimization framework includes the following optimization objectives.

For soft RTSs, we consider optimization of RRC with respect to AET. For the case of equidistant checkpointing, the optimization framework provides the optimal number of checkpoints resulting in the minimal AET. For non-equidistant checkpointing, the optimization framework provides two adaptive techniques that estimate the probability of errors and adjust the checkpointing scheme (the number of checkpoints over time) with the goal to minimize the AET.

While for soft RTSs analyses based on AET are sufficient, for hard RTSs it is more important to maximize the probability that deadlines are met. To evaluate to what extent a deadline is met, in this thesis we have used the statistical concept Level of Confidence (LoC). The LoC with respect to a given deadline defines the probability that a job (or a set of jobs) completes before the given deadline. As a metric, LoC is equally applicable for soft and hard RTSs. However, as an optimization objective LoC is used in hard RTSs. Therefore, for hard RTSs, we consider optimization of RRC with respect to LoC. For equidistant checkpointing, the optimization framework provides (1) for a single job, the optimal number of checkpoints resulting in the maximal LoC with respect to a given deadline, and (2) for a set of jobs running in a sequence and a global deadline, the optimization framework provides the number of checkpoints that should be assigned to each job such that the LoC with respect to the global deadline is maximized. For non-equidistant checkpointing, the optimization framework provides how a given number of checkpoints should be distributed such that the LoC with respect to a given deadline is maximized.

Since the specification of an RTS may have a reliability requirement such that all deadlines need to be met with some probability, in this thesis we have introduced the concept Guaranteed Completion Time which refers to a completion time such that the probability that a job completes within this time is at least equal to a given reliability requirement. The optimization framework includes Guaranteed Completion Time as an optimization objective, and with respect to the Guaranteed Completion Time, the framework provides the optimal number of checkpoints, while assuming equidistant checkpointing, that results in the minimal Guaranteed Completion Time.

Preface

This thesis summarizes my academic work carried out during the period between April 2009 and December 2014. During the period between April 2009 to December 2012, the research work was conducted in the Embedded System Laboratory group at the Department of Computer and Information Science, Linköping University, Sweden. Since January 2013 until present the research work has been conducted in the Digital ASIC group at the Department of Electrical and Information Technology, Lund University, Sweden.

The material presented in this thesis is based on the following publications.

BOOK CHAPTERS

- D. Nikolov, M. Väyrynen, U. Ingelsson, V. Singh, and E. Larsson, »Optimizing fault tolerance for multi-processor system-on-chip,« in *Design and Test Technology for Dependable Systems-on-chip*, R. Ubar, J. Raik & H. T. Vierhaus, Eds. Hershey: IGI Global, 2011, pp. 66-91.

  ☞ The research work has been carried out by both the first and the second author under the guidance of the remaining authors. The first author was the main integrator of the different parts presented in the book chapter.

JOURNAL ARTICLES

- D. Nikolov, U. Ingelsson, V. Singh, and E. Larsson, »Evaluation of level of confidence and optimization of roll-back recovery with checkpointing for real-time systems,« *Microelectronics Reliability*, vol. 54, no. 5, pp. 1022-1049, May 2014.

  ☞ The research work has been carried out by the first author under the guidance of the remaining authors.

- D. Nikolov, and E. Larsson, »Optimizing the Level of Confidence for Multiple Jobs,« submitted to *IEEE Transactions on Computers*.

  ☞ The research work has been carried out by the first author under the guidance of the second author.

PEER-REVIEWED CONFERENCE ARTICLES

- D. Nikolov, U. Ingelsson, V. Singh, and E. Larsson, »Estimating error-probability and its application for optimizing roll-back recovery with checkpointing,« in *Fifth IEEE International Symposium on Electronic Design, Test and Application, 2010. (DELTA'10)*, pp. 281-285, IEEE, 2010.

  ☞ The research work has been carried out by the first author under the guidance of the remaining authors.

- D. Nikolov, U. Ingelsson, V. Singh, and E. Larsson, »Level of confidence evaluation and its usage for roll-back recovery with checkpointing optimization,« in *IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W), 2011*, pp. 59-64, IEEE, 2011.

  ☞ The research work has been carried out by the first author under the guidance of the remaining authors.

- D. Nikolov, and E. Larsson, »Maximizing Level of Confidence for Non-Equidistant Checkpointing,« submitted to *IEEE European Test Symposium (ETS), 2015* .

  ☞ The research work has been carried out by the first author under the guidance of the second author.

The research work has been supported by:

- *European Union's 7th Framework Programme's collaborative research project FP7-ICT-2013-11-619871 BASTION - Board and SoC Test Instrumentation for Ageing and No Failure Found,*

- *European Union's 7th Framework Programme's collaborative research project FP7-2009-IST-4-248613 DIAMOND - Diagnosis, Error Modeling and Correction for Reliable Systems Design,*

- *Swedish Research Council (Vetenskapsrådet (VR)) Fault-tolerant design and optimization of multi-processor system-on-chip, Dnr:2009-4480,* and

- *The Swedish Foundation for International Cooperation in Research and Higher Education (STINT) Design of self-healing system chips, Dnr:YR2007-2008.*

Acknowledgments

It is funny how life goes by, sometimes you get elevated up in the clouds, and sometimes you get buried deep underground. After so many ups and downs, I am finally here, standing up, waiting for new challenges to rise up. Bring it on!

This thesis would have never seen the daylight without the help of some very important people to whom I would like to express my deepest gratitude.

My most sincere gratitude goes to my supervisor, Prof. Erik Larsson. First, I thank you Erik, for giving me the opportunity to pursue my graduate studies. Your constant optimism, encouragement, and confidence in this research work had become my steering wheel, and the force to complete this journey. Thank you for your patience and understanding, for inspiring me, and for helping me to improve my research skills. You have set an example worth following. Thank you very much for everything.

Further, I would like to thank the Head of the Electrical and Information Technology (EIT) Department at Lund University, Prof. Viktor Öwall, for letting me complete the studies here, and for all the efforts he puts to make our working environment at EIT very pleasant and enjoyable.

Many thanks to the administration staff for their kind help concerning administrative issues. In particular, I would like to thank Anne Andersson and Pia Bruhn for their help.

To my colleagues, former and present, members of the Digital ASIC group and the EIT department, thank you all for the positive atmosphere that you have created, and you keep on creating. Thank you for your support, for all the interesting discussions, and for the many joyful moments that have made this journey much easier.

I would like to acknowledge Farrokh Ghanni Zadegan and Breeta SenGupta who put substantial effort in proofreading and correcting the text of thesis.

You have incorporated fault tolerance into this thesis, literally.

During my studies, I spent half (and a bit more) of the time as a PhD student at the Department of Computer and Information Science (IDA) at Linköping University. With respect to this, I would like to thank the Head of the Department, Prof. Mariam Kamkar, for her help to make the transition from Linköping University to Lund University as smooth as possible. During the time I spent at Linköping University, I was part of the Embedded System Laboratory (ESLAB) group. Many thanks to Prof. Zebo Peng, the leader of ESLAB, for setting up a very positive, enjoyable, and welcoming atmosphere in ESLAB. To all my former colleagues, those who have already graduated and those who have not yet graduated, you have enriched my life with so many good memories. Thank you for your friendship, for the enjoyable discussions, and for taking the time to put up with me.

To my friends, thank you all for the tremendous support that you have given me. You have always been with me, sharing both the sad and the happy moments.

During the last couple of years, I have lost some very dear ones. I dedicate this thesis to them. You may have not lived for this moment, but you will always be a part of me, a part of this moment, a part of every moment.

To my brother, who has always been and will always be there for me, to share whatever this life brings with it.

To my nephew, for his smile and that simple phrase "*цицо Mume*" fill my heart with such a joy and happiness that it makes me forget about all the difficulties that I face over time.

Finally, to my parents... Whatever I say will not be enough to express my gratitude for all the support and the love you have given me throughout the years, and you keep on giving me. You always bring out the best in me.

Ви Благодарам!

Lund, December 2014

Contents

Popular Science Summary **vii**

Abstract **xi**

Preface **xiii**

Acknowledgments **xvii**

List of Figures **xxiii**

List of Tables **xxv**

1 Introduction **1**

1.1 Fault Tolerance 2

1.2 Real-Time Systems 4

1.3 Fault Tolerance in Real-Time Systems 5

1.4 Related Work 6

1.5 Thesis Scope 9

1.6 Thesis Contributions 11

1.7 Thesis Organization 13

2 Preliminaries **15**

2.1 System Model 15

2.2 Fault Model and Fault Assumptions 16

2.3 Definitions and Notations 17

I Equidistant Checkpointing **23**

3 Average Execution Time **27**
3.1 Problem Formulation 27
3.2 Calculating the Average Execution Time 28
3.3 The Optimal Number of Checkpoints 38
3.4 Experimental Results 43

4 Level of Confidence **47**
4.1 Single Job 48
4.1.1 Problem Formulation 48
4.1.2 Evaluation of the Level of Confidence 48
4.1.3 Properties of the Level of Confidence 54
4.1.4 Maximizing the Level of Confidence 55
4.1.5 Experimental Results 65
4.2 Multiple Jobs 70
4.2.1 Problem Formulation 71
4.2.2 Motivation 72
4.2.3 Exhaustive Search 77
4.2.4 Semi-Exhaustive Search 81
4.2.5 Experimental Results 92

5 Guaranteed Completion Time **99**
5.1 Problem Formulation 99
5.2 Definition and Properties 100
5.3 Minimizing Guaranteed Completion Time 108
5.3.1 Example 117
5.4 Experimental Results 121

6 Summary of Part I **125**

II Non-Equidistant Checkpointing **127**

7 Average Execution Time **131**
7.1 Problem Formulation 132
7.2 Motivation 132

7.3 Error Probability Estimation and Corresponding Adjustment . 134
7.4 Experimental Results 137

8 Level of Confidence 143
8.1 Problem Formulation 144
8.2 Motivation 144
8.3 Evaluation of Level of Confidence 145
8.4 Exhaustive Search 152
8.5 Clustered Checkpointing 157
8.5.1 Example 159
8.6 Experimental Results 163

9 Summary of Part II 175

III Thesis Summary 177

10 Conclusions and Future Work 179
10.1 Average Execution Time 180
10.2 Level of Confidence 181
10.3 Guaranteed Completion Time 182
10.4 Future Work 183

IV Appendix 185

Appendix A 187

Appendix B 193

Appendix C 213

Appendix D 219

References 225

List of Figures

2.1 System model 16
2.2 Illustration of checkpointing overhead 18
2.3 Graphical presentation of RRC scheme 19
2.4 Illustration of successful and erroneous execution segments . . 20

3.1 Detailed execution of a job employing RRC with n_c checkpoints 29
3.2 Illustration of possible outcomes when executing an execution segment 32
3.3 T_{ES}, average time that is spent only on execution of execution segments 36
3.4 T_{CO}, average time that is spent only on performing checkpointing operations 37
3.5 AET, average execution time of a job employing RRC 37

4.1 Number of cases $N_{n_c}(k)$ for $n_c = 3$ and $P_T = 0.5$ 50
4.2 Probability metric per case $P_\epsilon^{n_c} \times (1 - P_\epsilon)^k$ for $n_c = 3$ and $P_T = 0.5$ 51
4.3 Probability distribution function $p_{n_c}(k)$ for $n_c = 3$ and $P_T = 0.5$ 52
4.4 Illustration of models used for (a) stated problem formulation, (b) Local Optimization, and (c) Single Large Job approach . . . 73
4.5 Completion time 79

4.6 Completion time for checkpoint assignment $\bar{n}_c^{\dagger} = [3,4]$ 84
4.7 Completion time for checkpoint assignment $\bar{n}_c^{\dagger} = [4,3]$ 85
4.8 Flow chart of the Semi-Exhaustive Search method 91

5.1 Illustration of the completion time ${}^{n_c}t_k$ as a function of n_c, for given $T = 1000$ t.u. (time units), $\tau = 110$ t.u., and $k \in [0,5]$ . . 103
5.2 Illustration of ${}_{k^{\dagger}}^{k^{\ddagger}}n_{c_L}$ and ${}_{k^{\dagger}}^{k^{\ddagger}}n_{c_U}$, for given $T = 1000$ t.u., $\tau = 110$ t.u., and $k \in [1,5]$ 107
5.3 Illustration of GCT_{δ} and ${}_{k}^{g}n_c$, for given $T = 1000$ t.u., $\tau = 110$ t.u., $P_T = 0.9$, $\delta = 0.99999998$, and $k \in [5,10]$ 109
5.4 Flow chart of presented method for minimizing GCT_{δ} 110
5.4 Flow chart of presented method for minimizing GCT_{δ} (*continued*) 111
5.5 Search algorithm for obtaining ${}_{k}^{g}n_c$ 115
5.6 Illustration on finding the minimal GCT_{δ}, for given $T = 1000$ t.u., $\tau = 20$ t.u., $P_T = 0.5$, and $\delta = 1 - 10^{-18}$ 118

7.1 Impact of inaccurate error probability estimation on AET in terms of relative deviation from the optimal AET (%) 134
7.2 Graphical presentation of PPE 135
7.3 Graphical presentation of APE 136
7.4 Relative deviation from the optimal AET (%) for a constant *real* error probability $Q_T = 0.01$ 139
7.5 Illustration of the three error probability profiles (a) $Q_1(t)$, (b) $Q_2(t)$, and (c) $Q_3(t)$ 141

8.1 Flow chart of the Clustered Checkpointing method 158
8.2 Distributions explored by the CC method for the given inputs $L = 3$, $T = 24$ t.u., and $n_c = 5$ 161

List of Tables

3.1 Input scenarios 43
3.2 AET obtained for $n_c \in [1,20]$, for Scenario A 44
3.3 AET obtained for $n_c \in [1,20]$, for Scenario B 44
3.4 AET obtained for $n_c \in [1,20]$, for Scenario C 45
3.5 AET obtained for $n_c \in [1,20]$, for Scenario D 45

4.1 Input scenarios 65
4.2 $\Lambda_{n_c}(D)$ for different n_c values, for Scenario A 67
4.3 $\Lambda_{n_c}(D)$ for different n_c values, for Scenario B 67
4.4 Input scenarios 93
4.5 Comparison of $\bar{\lambda}_{\vec{n}_c^*}(D)$ for the Local Optimization (LO), the Single Large Job (SLJ), the Exhaustive Search (EXS), and the Semi-Exhaustive Search (SES) approach 95
4.6 Comparison of the computation time for the Exhaustive Search (EXS) and the Semi-Exhaustive Search (SES) approach 97

5.1 Input scenarios 122
5.2 GCT_δ and the number of re-executions k included in GCT_δ, for Scenario A, at various number of checkpoints n_c 122

5.3 GCT_δ and the number of re-executions k included in GCT_δ, for Scenario B, at various number of checkpoints n_c 123

7.1 Three error probability profiles: $Q_1(t)$, $Q_2(t)$, and $Q_3(t)$ 139
7.2 Relative deviation from the processing time T (%) for the three error probability profiles $Q_1(t)$, $Q_2(t)$, and $Q_3(t)$ 140

8.1 Number of *all* possible distributions of n_c checkpoints in a job with a processing time $T = 1000$ t.u. 154
8.2 Number of distributions of n_c checkpoints, explored with the EXS method, for a job with a processing time $T = 1000$ t.u. . . 157
8.3 Input scenarios 163
8.4 Comparison of $\bar{\Lambda}_{n_c}(D)$ for different checkpointing schemes at different n_c values for Scenario A 166
8.5 Comparison of $\bar{\Lambda}_{n_c}(D)$ for different checkpointing schemes at different n_c values for Scenario B 167
8.6 Comparison of $\bar{\Lambda}_{n_c}(D)$ for different checkpointing schemes at different n_c values for Scenario C 168
8.7 Comparison of $\bar{\Lambda}_{n_c}(D)$ for different checkpointing schemes at different n_c values for Scenario A 171
8.8 Comparison of $\bar{\Lambda}_{n_c}(D)$ for different checkpointing schemes at different n_c values for Scenario B 173
8.9 Comparison of $\bar{\Lambda}_{n_c}(D)$ for different checkpointing schemes at different n_c values for Scenario C 174

1

Introduction

The constant demand for high performance has resulted in a rapid development of semiconductor technologies. The development mainly relates to the continuous shrinking of the transistor sizes reaching the sub-micron domain where the size of the transistors is measured in nanometers. Such small transistor sizes offer the possibility of a very large scale integration process which enables fabrication of very complex integrated circuits (ICs). With such ICs it is possible to integrate an entire system onto a single chip commonly referred to as a System on Chip (SoC). To further improve performance, an SoC is often designed to include multiple processors, and it is referred to as a Multi-Processor SoC (MPSoC).

While the latest semiconductor technologies offer previously unseen performance, there is a drawback that comes along as we step into the sub-micron domain. Shrinking feature sizes and lowering operation voltages make devices more susceptible to soft errors [1], [2], [3], [4], [5]. Soft errors occur as a result of a particular class of hardware faults known as transient faults [6], [7]. These faults have a short lifetime, *i.e.* they appear in the system, but disappear after a short period of time, and usually they are caused by different external factors, such as high-energy particle hits originating from different radiation sources including cosmic rays, electromagnetic interference, etc. [8], [9], [10]. Despite the short lifetime of the transient faults, these faults often result in soft errors. Taking no actions when soft errors are present in the system may lead to a system failure. Thus, soft errors can significantly influence the system reliability.

The soft error rate observed in recent technologies has increased by orders of magnitude compared to earlier technologies, and the rate is expected to grow in future semiconductor technologies [5], [11], [12], [13], [14]. Therefore, it is becoming increasingly important to consider techniques that enable detection and recovery from soft errors [11], [15], [16], [17].

1.1. FAULT TOLERANCE

Fault tolerance is the property that enables a system to continue with its correct operation even in the presence of faults (errors), and it is generally implemented by error detection and subsequent system recovery [18], [7], [19], [20]. Fault tolerance has been a subject of research for a long time, and significant amount of work has been produced over the years [18], [21], [22], [23], [24], [25], [26]. To provide fault tolerance, systems are usually designed such that some redundancy is included. The common types of redundancy used are information, hardware, and time redundancy.

Error-detecting and error-correcting codes provide fault tolerance while using information redundancy, *i.e.* the data includes additional information (check bits) that can verify the correctness of the data before it is used (error-detection), or even correct erroneous data bits (error-correction). Different error-detecting and error-correcting codes have been proposed including parity codes, cyclic codes, arithmetic codes etc. [27], [28], [29]. The major disadvantage of error-detecting and error-correcting codes is that they are limited to errors that occur during transfer of data (system bus) or errors in memory.

Employing hardware redundancy is a very common practice to provide fault tolerance. Already in 1952, John von Neumann introduced a redundancy technique called NAND multiplexing for constructing reliable computation from unreliable devices [30]. N-Modular Redundancy (NMR) is another method that provides fault tolerance at the cost of adding N hardware replicas. The correct output in an NMR is obtained by voting on the outputs generated from the N modules. An example of providing fault tolerance by adding redundant hardware is the architecture of the fighter JAS 39 Gripen which contains seven hardware replicas [31]. The most common form of NMR is Triple Modular Redundancy (TMR) [7], [32], [33], [34], [35]. In TMR, a process is executed on three functionally equivalent hardware modules, and the output is obtained by using the two-out-of-three voting concept. Thus, even if one of the hardware units in TMR fails, the system is still able to produce the correct output. Multiple variations of NMR exist. Unit-level modular redundancy applies the same concept as NMR at a higher granularity, *i.e.* instead of only replicating entire modules, it replicates also units (subsystems) within the modules [7], [36]. Dynamic redundancy is another variation of NMR, but the major difference is that even though there are N replicated modules, only one module is active at a time, and in case of errors, the active module is replaced by a spare module [37]. Hybrid redundancy requires even more hardware redundancy because it assumes that all N module are active at a time, and in case errors are detected in some modules, the erroneous modules are replaced by spare modules [38]. In hybrid redundancy, the erroneous modules are detected by comparing the output of the voter with the outputs of the

active modules, and all the active modules which are in disagreement with the voter need to be replaced by spare modules. Sift-out modular redundancy is another variation of NMR, but instead of using a majority voter, this technique uses comparator, detector, and collector circuits, where the comparator and the detector circuits are used to exclude the erroneous modules from the system, and the collector circuit is used to provide the system output [39].

The major advantage of NMR, and any of the similar techniques discussed earlier, is that it is capable of error-masking (disabling propagation of errors), as long as less than half of the modules are affected by errors. Still, if more than half of the modules are effected by errors, NMR is able to detect errors, and in such case a system recovery needs to be performed. The major disadvantage of NMR is that it is a costly solution to implement fault tolerance because it requires substantial amount of hardware redundancy.

Time redundancy is another way to provide fault tolerance. However, fault tolerance techniques that use time redundancy are only efficient if the faults are of transient nature, *i.e.* faults that occur, but disappear after a short period of time. These transient faults often result in soft errors. The simplest technique that uses time redundancy copes with soft errors by executing the same program twice, and it obtains the correct result if the outputs of the two executions match. Roll-back Recovery with Checkpointing (RRC) is a well-known fault tolerance technique that efficiently copes with soft errors. RRC has been the focus of research for a long time [40], [41], [42], [43], [44], [45]. Unlike classical re-execution schemes where the task (job) is restarted once an error is detected, RRC copes with soft errors by making use of previously stored error-free states of the task, referred to as checkpoints. During the execution of a task, the task is interrupted and a checkpoint is taken and stored in a memory. The checkpoint contains enough information such that a task can easily resume its execution from that particular point. For RRC it is crucial that each checkpoint is error-free, and this can be done by for example running acceptance tests to validate the correctness of the checkpoint. Once the checkpoint is stored in memory, the task continues with its execution. As soft errors may occur at any time during the execution of a task, an error detection mechanism is used to detect the presence of soft errors. There are various error detection mechanisms that can be used, *e.g.* watchdogs, duplication schemes etc. [7], [46], [47], [48]. In case that the error detection mechanism detects an error, it forces the task to roll-back to the latest checkpoint that has been stored.

Depending on the implementation, different RRC schemes exist, and they differ among each other based on the following two key aspects. The first aspect is how much information is stored at each checkpoint. With respect to this, there are two different RRC schemes, *i.e.* full checkpointing [49], [50], [51], and incremental checkpointing [52], [53], [54]. In a full checkpointing

scheme, at each checkpoint the complete state of the job is stored, while in an incremental checkpointing scheme only the changes with respect to the most recent saved state are stored.

The second key aspect relates to when checkpoints are taken. With respect to this, there are two different RRC schemes, *i.e.* equidistant checkpointing [55], [51], [50], [56], [57] and non-equidistant checkpointing scheme [58], [59], [60], [61] . In equidistant checkpointing, the checkpoints are distributed evenly throughout the execution of the job (meaning that the distance between two successive checkpoints is always the same), while in non-equidistant checkpointing, the checkpoints are not evenly distributed throughout the execution of the job (the distance between two successive checkpoints is not always the same).

As shown in this section, employing fault tolerance is often related to adding an overhead which can result in: higher hardware cost, higher energy consumption, and even affect (degrade) system's performance. Therefore, there should be a clear goal to what extent fault tolerance is required for a particular system. Minimizing the drawback caused by employing fault tolerance usually requires optimization of the fault tolerance technique which is used. The optimization goals for a given fault tolerance technique may differ among the different classes of computer systems where fault tolerance is employed. In general, computer systems are classified into non-real-time and real-time systems depending on the requirement to meet a given time constraint.

1.2. REAL-TIME SYSTEMS

A real-time system (RTS) is a computer system where the correctness of the system behavior depends not only on the logical result of the computations, but also on the physical time when these results are produced [62]. Thus, in RTSs there exists a requirement of meeting time constraints (deadlines). These systems are used in various domains including digital control, signal processing, telecommunication, medical systems etc. Depending on the consequences when the deadlines are violated, RTSs are often classified into soft and hard RTSs [62]. For hard RTSs, it is a catastrophe if deadlines are not met, while for soft RTSs, violating the deadlines usually degrades the quality of service, but the consequences are not catastrophic [63]. An example of a hard RTS can be the control system in an airplane, where the consequences of violating the deadlines can be catastrophic. On the other hand, an example of a soft RTS can be a system that provides a video streaming service, where missing a deadline does not cause catastrophic damage to the system, but affects the performance of the system negatively.

To determine if an RTS meets its time constraints, schedulability analysis is performed during the early design stage, *i.e.* before the system is deployed (implemented). The schedulability analysis provides the answer to the question if all jobs are able to meet their corresponding deadlines. During the design of a soft RTS it is common to perform schedulability analysis based on the average execution time (AET), while for a hard RTS it is common during the design to perform schedulability analysis based on the worst case execution time (WCET) to ensure that deadlines are met.

Since for soft RTSs the schedulability analysis is based on AET, there is a motivation to minimize the AET. Obtaining the minimal AET has some important advantages. For example, reducing the AET can lead to a lower power consumption, and thus can save energy which is very important for systems with limited power budget (the lifetime of many embedded RTSs depends on a battery source). Minimizing the AET of a job can improve the system's throughput, *i.e.* the number of jobs executed over time, and by this improve performance. For soft real-time control systems, where tasks are executed periodically, reducing the AET is important as it can affect the control quality, *i.e.* tasks' periods can be adjusted according to AET estimates while optimizing a control performance criterion [64]. Despite the advantages of minimizing the AET, the main drawback with AET is that it does not guarantee that deadlines are met, and it lacks the distribution of the execution times, *i.e.* sometimes a task (job) can complete much earlier in time, while sometimes it may take much longer time.

For hard RTSs the schedulability analysis is based on WCET. Using WCET guarantees that the deadlines are always met, and thus catastrophic consequences are avoided. However, it is difficult to accurately estimate the WCET which is a deterministic upper bound of the time required for a job to complete [65], [66]. Furthermore, the WCET may be very pessimistic which can lead to increased cost due to having over-designed systems, *i.e.* systems that are equipped with more resources than normally needed to perform the intended function.

1.3. FAULT TOLERANCE IN REAL-TIME SYSTEMS

Since RTSs, like any other computer system manufactured in the latest semiconductor technologies, are susceptible to soft errors, it is important to employ fault tolerance in RTSs as well. However, fault tolerance usually introduces a time overhead which negatively impacts RTSs. The time overhead due to usage of fault tolerance may increase the AET, and it may lead to a missed deadline. Therefore, special consideration should be taken when employing fault tolerance in RTSs.

The following studies have addressed usage of fault tolerance in RTSs. Punnekkat *et al.* performed schedulability analysis while assuming that a fault can adversely affect only one job at a time [67]. Kandasamy *et al.* considered a fault model which assumes that only one single transient fault may occur in any of the nodes during the execution of an application [68]. This model has been generalized to address a number k of transient faults in the work of Pop *et al.* [69]. Researchers have shown that the schedulability of an application can be guaranteed for preemptive on-line scheduling, where a low priority task can be interrupted by a higher priority task, under the presence of a single transient fault [70], [71], [72], [73], [74].

The studies presented earlier, have considered schedulability analysis based on WCET while considering that the number of faults (errors) is bounded to a fixed number. For each error that occurs in the system, the employed fault tolerance technique performs the error recovery step, and by doing so it increases the execution time of the job, *i.e.* the time required for the job to complete. When the number of errors is bounded to a fixed number, it is possible to estimate the WCET by using the maximum number of errors that can occur, given that the time overhead introduced due to usage of fault tolerance is deterministic (it is possible to estimate the upper bound of the time it takes to perform the error recovery step). However, faults can occur unexpectedly at any moment in time with some probability. In such case, it is not possible to estimate the WCET. Instead, when errors occur with some probability, there is a probability the deadlines might be missed. Therefore, when optimizing the usage of fault tolerance in RTSs, another important aspect to consider is optimization of fault tolerance such that the probability to meet the deadlines is maximized, which is needed for hard RTSs, or at least the probability to meet the deadlines is higher than a given reliability requirement, which may be needed for soft RTSs.

1.4. RELATED WORK

In this section, we discuss related work in relation to the scope of this thesis. Therefore, we discuss studies that address RRC in RTSs. As mentioned earlier, the major drawback of RRC is the introduced time overhead that depends on the number of checkpoints. The time overhead introduced by RRC impacts soft and hard RTSs differently. For soft RTSs the time overhead increases the AET and thus, degrades system's performance, while for hard RTSs the time overhead may be the reason that the time constraints (deadlines) are violated. Therefore, to minimize the negative impact caused by the introduced time overhead, it is important to consider optimization of RRC. Next, we discuss the different optimization objectives that have been considered for RRC in soft and hard RTSs.

For soft RTSs, most of the work addressing RRC aims to optimize the RRC scheme with the goal to minimize the AET [47], [50], [75], [61], [76], [57]. RRC has an impact on the AET because the number of checkpoints that is taken during the execution of a job affects the execution time. A high number of checkpoints reduces the time overhead that is caused due to re-execution. However, a high number of checkpoints increases the time overhead that is caused due to taking the checkpoints. On the other hand, when a low number of checkpoints is used, the time overhead of taking the checkpoints is lower, but then the time overhead caused due to re-execution is higher. From this discussion, we conclude that there exists a trade-off when selecting the number checkpoints to be used. Such trade-off motivates the existence of an optimal number of checkpoints that minimizes the time overhead, and therefore minimizes the AET.

Most studies assume that the fault-free execution (computation) time is given [50], [75], [76], [57]. Ziv *et al.* have presented a technique to analyze the AET of four different checkpointing schemes based on task duplication [47]. Nakagawa *et al.* have analyzed the AET for three different checkpointing schemes based on double modular redundancy and derived analytical expressions for the optimal checkpoint intervals [75]. Common assumption in these studies is that the checkpoints are evenly distributed throughout the execution of the job (equidistant checkpointing). Analyses on AET while assuming that the checkpoints are not evenly distributed are presented in [61], [50].

Shin *et al.* derive analytical expressions for calculating the AET for two different models: (1) basic model, where they use the assumption of perfect coverage of the on-line detection mechanisms, and (2) extended model, where they use the assumption of imperfect coverage of the on-line detection mechanisms and the acceptance tests [50]. For the basic model, due to the assumption of perfect coverage of the on-line detection mechanisms, whenever a job completes it always provides the correct results. For the extended model, due to the assumption of imperfect coverage of the on-line detection mechanisms and the acceptance tests, a job may complete with an unreliable (incorrect) result. In [50], for the basic model, the authors obtain the optimal number of checkpoints that results in the minimal AET. For the extended model, the authors provide an algorithm to obtain the optimal placement of checkpoints that minimizes the AET while the probability of an unreliable result is kept below a specified level. An important conclusion from their work is that for the basic model, the minimal AET is achieved while using equidistant checkpointing. However, that is not necessarily the case for the extended model. Similar conclusions to this have been presented by Ziv *et al.* [61].

For hard RTSs, most work addressing RRC focuses on providing guarantees that time constraints are met [70], [72], [56], [51], [77], [78], [79]. This is done either by analyzing the WCET when RRC is used, or by analyzing the

probability to meet the deadline.

The following studies have analyzed RRC with respect to worst case execution time (WCET) [56], [69], [67]. Zhang *et al.* discuss fault recovery based on checkpointing for hard RTSs [56]. In their work, the authors assume a hard RTS that executes a set of n periodic real-time jobs, where each of the jobs is modeled with three parameters: execution time under fault-free conditions, period and deadline. For the given system they assume two different fault models: (1) at most k faults can occur during the execution of a single job, and (2) at most k faults can occur during a hyper-period (shortest repetitive sequence of the schedule). To handle faults they assume that each job employs checkpointing. The main contribution of their work is providing schedulability tests to verify if a given hard RTS is schedulable, *i.e.* all jobs are periodically executed and able to meet the given deadlines, under the given fault model. If a system is schedulable under the given fault model, they report the required checkpointing scheme, *i.e.* the number of checkpoints to be used for each job. The schedulability analysis provided in this work is based on calculating the response time (WCET). They show that employing checkpointing is very important to obtain schedulability for hard RTSs. Zhang *et al.* have shown that a system that is schedulable under a given fault model when checkpointing is employed, may not be schedulable under the same fault model if re-execution is used instead of checkpointing [56]. The real-time guarantees that are provided in this work rely on the response time (WCET) analysis. The main drawback with WCET is that accurate estimate of WCET is possible only for a fault model where the number of faults is bounded to a fixed number. However, due to the fact that errors can occur at any moment in time, it is difficult to predict the number of faults that can occur within an interval of time.

When RRC is employed in hard RTS, the number of checkpoints used affects the probability to meet the deadlines [51], [78], [79]. Kwak *et al.* provide analysis on the reliability of a checkpointed real-time control systems [51]. In their work, the authors consider a control system that consists of a single control task for which the WCET, the period, and the deadline are given. For the fault model, they consider that transient faults occur according to a Poisson process with a fault arrival rate λ, and recovery rate μ. By utilizing Markov models, they derive the reliability equation over a mission time (number of consecutive sampling periods) of the control system. Kwak *et al.* model the control system by a 3-state Markov chain, where within a sampling period the control system can be in one of the following states: (1) the control task has been correctly executed and the transient faults are in a *fault-free state*, (2) the control task has been correctly executed and the transient faults are in a *fault-active state* and (3) the control task has either been executed incorrectly or has not finished within the sampling period. Kwak *et al.* have shown the impact of the number of checkpoints on the system reliability, and therefore they

have proposed an algorithm to find the optimal number of checkpoints with the goal to maximize system reliability [51]. Further, they extend the model to address system reliability for a control system which consists of multiple control tasks, where for each control task given are the WCET, the period, and the deadline D_i which is equal to the period. For the case of multiple control tasks, they propose a task allocation algorithm. The algorithm computes the greatest common divisor of deadlines ($GCDD$), and divides each control task into $D_i/GCDD$ equal length smaller chunks (subtasks). Each subtask is then sequentially assigned on each $GCDD$ interval. After running the task allocation algorithm, they apply the system reliability model for a single control task with a deadline equal to $GCDD$, assuming that all the subtasks assigned within the $GCDD$ interval are equivalent to one control task. Out of this, they can obtain the optimal number of checkpoints to be used for each task such that the system reliability is maximized. In another study, Kwak *et al.* discuss multiple real-time tasks and derive an explicit formula of the probability that all tasks are successfully completed with a given set of checkpoint intervals [78]. Using Markov model, Kwak *et al.* calculate the probability of task completion against faults that occur in a Poisson process for a checkpoint scheme [79].

The conclusion from the presented related work regarding RRC is that minimizing the AET is important for soft RTSs, while maximizing the probability to meet deadlines is important for hard RTSs.

1.5. THESIS SCOPE

The scope of this thesis is employing fault tolerance in RTSs in order to cope with soft errors which are caused by transient faults. Important to note is that while fault tolerance is not limited only to transient faults, in this thesis we only address soft errors that occur as a result of transient faults. For that reason, we consider Roll-back Recovery with Checkpointing (RRC) which is a well-known fault tolerance technique that efficiently copes with soft errors. The major drawback of RRC is that it introduces a time overhead which negatively affects RTSs. The goal of this thesis is to provide an optimization framework for RRC when used in RTSs while considering different optimization objectives which are important for RTSs.

In this thesis, we consider a scheme for RRC that utilizes task duplication. In such scenario, a task (job) is duplicated and concurrently executed on two processing nodes. During the execution of the job a number of checkpoints are taken. At each checkpoint the states of both processing nodes are compared against each other. If the states match, the states are saved as a safe point (correct checkpoint) from which the job can be resumed. If the states do

not match, this indicates that an error has occurred in at least one of the processing nodes, and to recover from the error, both processing nodes have to load the latest saved safe state, and resume the execution of the job from that point. By using this scheme we avoid the usage of acceptance tests to verify the correctness of the checkpoints at the cost that errors can only be detected at discrete time points, *i.e* when comparing the checkpoints from the two processing nodes. Depending on how the checkpoints are distributed over the execution of the job, we explore the two checkpointing schemes: equidistant and non-equidistant checkpointing. For the considered RRC schemes we provide an optimization framework which includes multiple optimization objectives.

The purpose of such an optimization framework is to assist the designer of an RTS during the early design stage, when the designer needs to explore different fault tolerance techniques and choose a particular fault tolerance technique that meets the specification requirements for the RTS that is to be implemented. The designer of an RTS can use this optimization framework to acquire knowledge if RRC is a suitable fault tolerance technique for the RTS which needs to be implemented. Next, we discuss the different optimization objectives for RRC that are considered in this thesis.

For soft RTSs, we consider optimization of RRC with respect to AET. For the case of equidistant checkpointing, the optimization framework provides the optimal number of checkpoints resulting in the minimal AET. For non-equidistant checkpointing, the optimization framework provides two adaptive techniques that estimate the probability of errors and adjust the checkpointing scheme (the number of checkpoints over time) with the goal to minimize the AET.

While for soft RTSs analyses based on AET are sufficient, for hard RTSs it is more important to maximize the probability that deadlines are met. To evaluate to what extent a deadline is met, in this thesis we have used the statistical concept Level of Confidence (LoC). The LoC with respect to a given deadline defines the probability that a job (or a set of jobs) completes before the given deadline. As a metric, LoC is equally applicable for soft and hard RTSs. However, as an optimization objective LoC is used in hard RTSs. Therefore, for hard RTSs, we consider optimization of RRC with respect to LoC. For equidistant checkpointing, the optimization framework provides (1) for a single job, the optimal number of checkpoints resulting in the maximal LoC with respect to a given deadline, and (2) for a set of jobs running in a sequence and a global deadline, the optimization framework provides the number of checkpoints that should be assigned to each job such that the LoC with respect to the global deadline is maximized. For non-equidistant checkpointing, the optimization framework provides how a given number of checkpoints should be distributed such that the LoC with respect to a given deadline is maximized.

Since the specification of an RTS may have a reliability requirement such

that all deadlines need to be met with some probability, in this thesis we have introduced the concept Guaranteed Completion Time which refers to a completion time such that the probability that a job completes within this time is at least equal to a given reliability requirement. The optimization framework includes Guaranteed Completion Time as an optimization objective, and with respect to the Guaranteed Completion Time, the framework provides the optimal number of checkpoints, while assuming equidistant checkpointing, resulting in the minimal Guaranteed Completion Time.

In the next section, we detail the contributions of this thesis.

1.6. THESIS CONTRIBUTIONS

This section summarizes the contributions of this thesis. The contributions are divided into two parts. First, we present our contributions with respect to equidistant checkpointing and second, we present our contributions with respect to non-equidistant checkpointing.

The contributions with respect to equidistant checkpointing are divided into three groups based on the different optimization objectives. The optimization objectives considered for equidistant checkpointing are: *Average Execution Time*, *Level of Confidence* and *Guaranteed Completion Time*.

The contributions with respect to *Average Execution Time* are as follows:

- we derive a mathematical framework to evaluate the average execution time (AET) of a job when RRC is applied with a number of checkpoints;
- we derive a closed-form mathematical expression to compute the optimal number of checkpoints such that the minimal AET is obtained;

The contributions with respect to *Level of Confidence* are divided into two groups, *i.e.* (1) Level of Confidence (LoC) for a single job and (2) LoC for multiple jobs. With respect to LoC for a single job, the contributions are as follows:

- we derive an expression to evaluate the LoC with respect to a given deadline when RRC is employed with a number of checkpoints;
- we provide theorems along with mathematical proofs to prove specific properties of the expression used for evaluation of the LoC;
- we present a method, based on the presented theorems, to identify the optimal number of checkpoints that results in the maximal LoC;
- we show that the optimal number of checkpoints that results in the minimal AET, does not provide the maximal LoC.

With respect to LoC for multiple jobs, the contributions are as follows:

- we show that performing a local optimization for each job and combining these local optima together does not result in the maximal LoC with respect to the given global deadline;
- we show that handling the set of jobs as one single large job and obtaining the optimal number of checkpoints for the single large job does not result in the maximal LoC with respect to the given global deadline;
- we provide an expression to evaluate the LoC with respect to a given global deadline for a given checkpoint assignment, where the checkpoint assignment defines the number of checkpoints to be used by each job in the given set of jobs;
- we show that a holistic solution (exhaustive search on possible checkpoint assignments) is required to obtain the optimal checkpoint assignment and the maximal LoC with respect to the given global deadline;
- we propose a method (heuristic) to speed up the computations and obtain the results in significantly shorter time compared to the exhaustive search method.
- we present experimental results to demonstrate that our method is capable of finding the optimal checkpoint assignment and the maximal LoC while observing tremendous reduction in computation time compared to the exhaustive search method.

The contributions with respect to *Guaranteed Completion Time* are as follows:

- we bridge the gap between soft and hard RTSs by introducing the term Guaranteed Completion Time GCT_δ which takes into consideration both completion time and an LoC requirement;
- we propose an optimization method that finds the optimal number of checkpoints that results in the minimal GCT_δ, *i.e.* the minimal completion time that satisfies a given LoC requirement δ;

The contributions with respect to non-equidistant checkpointing are divided into two groups based on the different optimization objectives. The two optimization objectives considered for non-equidistant checkpointing are: *Average Execution Time* and *Level of Confidence*.

The contributions with respect to *Average Execution Time* are based on the assumption that the error probability may not be known at design time and it may change during operation. In particular, these are the contributions with respect to *Average Execution Time*:

- we study the impact of inaccurate error probability estimates on the AET;
- we present two techniques, namely Periodic Probability Estimation and Aperiodic Probability Estimation, to estimate the error probability and adjust RRC during runtime such that the AET is reduced.

The contributions with respect to *Level of Confidence* when non-equidistant checkpointing is used are as follows:

- we study the impact of non-equidistant checkpointing on the LoC with respect to a given deadline;
- we derive a mathematical expression to evaluate the LoC with respect to a given deadline for a given distribution of a number of checkpoints;
- we analyze an exhaustive search method to get a better understanding on how to find the optimal distribution of a given number of checkpoints that maximizes the LoC;
- we propose a method (heuristic) that aims to find the optimal distribution of a given number of checkpoints that maximizes the LoC at significantly lower complexity than the exhaustive search method;
- we present experimental results which show that the proposed method is capable of finding the optimal distribution of a given number of checkpoints that results in the maximal LoC with respect to a given deadline.

1.7. THESIS ORGANIZATION

The rest of the thesis is organized as follows. **Chapter** 2 details the common assumptions that are used throughout the thesis. Important definitions and notations are also defined and detailed in this chapter.

In **Part I**, we discuss optimization of RRC while assuming equidistant checkpointing. **Part I** consists of four chapters, namely **Chapter 3–6**. **Chapter 3** discusses RRC optimization with respect to AET. The steps of deriving the mathematical framework used to compute the optimal number of checkpoints that results in the minimal AET are covered in **Chapter 3**. **Chapter 4** discusses RRC optimization with respect to LoC. Two models are considered in **Chapter 4**, *i.e.* single job and multiple jobs. For the single job model, derivation of the expression for evaluation of the LoC with respect to a given deadline is covered and a method for finding the optimal number of checkpoints that

maximizes the LoC is presented. For the multiple jobs model, it is shown that the findings presented for the single job model are not directly applicable for the multiple jobs model. Therefore, a new expression for evaluation of the LoC is derived, and a method that finds the optimal assignment of the number of checkpoints for each job that maximizes the LoC is presented. **Chapter 5** discusses RRC optimization with respect to the guaranteed completion time GCT_δ. Definition and properties of GCT_δ along with an optimization method that finds the optimal number of checkpoints that minimizes the GCT_δ is presented in this chapter. **Chapter 6** summarizes **Part I**.

In **Part II**, optimization of RRC while assuming non-equidistant checkpointing is discussed. **Part II** consists of three chapters, namely **Chapter 7–9**. **Chapter 7** focuses on optimization of RRC with respect to AET. Two approaches that estimate the error probability over time (during the execution of the job), and based on the estimates adjust the checkpointing scheme with the goal to reduce the AET are presented in **Chapter 7**. **Chapter 8** studies the impact of non-equidistant checkpointing on the LoC. Mathematical expression for evaluation of the LoC with respect to a given deadline is derived, and an optimization method that aims to find the optimal distribution of a given number of checkpoints that maximizes the LoC is presented. **Chapter 9** summarizes **Part II**.

Finally, **Part III** concludes the thesis where in **Chapter 10** conclusions and future work are presented.

2

Preliminaries

This chapter covers the preliminary concepts and assumptions that are used throughout the thesis. The chapter is organized in three sections. First, we provide the system model, *i.e.* the architecture of a system that enables the usage of RRC. Second, we present the fault model and we present the fault assumptions regarding occurrence of soft errors. Finally, we define some key terms and notations that are later used in the thesis.

2.1. SYSTEM MODEL

The system model is presented in Figure 2.1. The architecture shown in Figure 2.1 consists of two processing nodes (processors), a shared memory and a Compare & Control Unit (CCU) connected through a shared bus. In such architecture RRC is performed as follows. Each job is duplicated and concurrently executed on both processing nodes. At a given time (a checkpoint request), the execution of the job is interrupted and a checkpoint is taken at each node. The checkpoint includes sufficient information such that the job can be resumed from that particular point. We consider a checkpoint to be represented as the state (status) of a processing node. Once the states of both nodes are obtained, each processing node sends its state to the CCU. The CCU compares the states from both processing nodes. If the states match, *i.e.* no errors are detected, the CCU stores one of the states in memory and signals to the processing nodes to continue with the execution of the job. If the states do not match, *i.e.* an error is detected, the CCU loads the most recently saved state from memory and sends it to both processing nodes forcing them to roll-back the execution of the job.

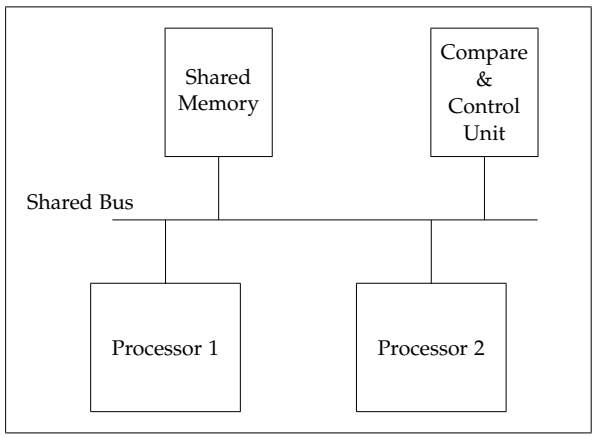


Figure 2.1.: System model

2.2. FAULT MODEL AND FAULT ASSUMPTIONS

For the fault model, we consider that soft errors (faults) that occur in the processing nodes cause erroneous outcome of the undergoing computation, *i.e.* bit-flips in the result produced after some computation. The fault model considers the occurrence of soft errors as an independent event. This means that the occurrence of a soft error does not depend on previous soft errors that have occurred. Further, the fault model considers that the probability P_t that no errors occur in a processing node within an interval of length t is given. This model is not limited to the number of faults that can occur within a time interval, which is an assumption that has been used in other research studies [67], [56], [77].

Due to the fact that the occurrence of soft errors is an independent event, the model allows to compute the probability P_τ that no errors occur within any interval of length τ, by using the following expression:

$$P_\tau = P_t^{\frac{\tau}{t}} \tag{2.1}$$

Observe in Eq. (2.1) that to evaluate P_τ it is necessary that P_t is provided, where P_t represents the probability that no errors occur in a processing node within an interval of length t.

Next we elaborate on the fault assumptions that are used in the thesis.

While soft errors can occur in any part of a computer system, *i.e.* memories, communication controllers, buses, etc., in this thesis, we address soft errors that occur only in the processing nodes, and we assume that errors occurring

elsewhere in the system are handled with conventional techniques for fault tolerance, *e.g.* error-correction codes (ECC) for handling soft errors that occur in memory. Further, we assume that each soft error provides a unique erroneous outcome. By using this assumption, if two soft errors occur, one in each processing node, the states of both processing nodes will differ due to that each soft error has caused a different erroneous outcome.

2.3. DEFINITIONS AND NOTATIONS

In this section, we define some useful terms related to RRC.

Checkpoint setup overhead is defined as the time needed for a processing node to prepare the contents of a checkpoint that is to be taken. In other words, the checkpoint setup overhead represents the time needed to generate a checkpoint which usually involves operations such as extracting the values of all registers in a processing node. We denote the checkpoint setup overhead with τ_s.

Bus communication overhead is defined as the time needed for a checkpoint to be transferred over the shared bus. A checkpoint can be either transferred from a processing node to the CCU, or from the CCU to one of the processing nodes. Since in the system model a shared bus is used, only one processing node at a time can exchange checkpoints with the CCU. We denote the bus communication overhead with τ_b.

Comparison overhead is defined as the time needed for the CCU to compare the checkpoints received from the processing nodes. After comparison, the CCU sends the appropriate checkpoint to both processing nodes. In case the checkpoints from both processing nodes match, the checkpoint is stored in memory and it is sent to both nodes. In case the checkpoints from the processing nodes do not match, the CCU retrieves the most recently stored checkpoint from memory and sends it to both nodes. Observe that the operations done by the CCU to load or store a checkpoint in memory are considered as a part of the comparison overhead. We denote the comparison overhead with τ_c.

Checkpoint unload overhead is defined as the time needed to extract (unload) the information from a checkpoint into the registers of a processing node. This overhead occurs after both processing nodes have received the checkpoint sent from the CCU. We denote the checkpoint unload overhead with τ_u.

Finally, we define the **checkpointing overhead** as the total time that is needed to perform all the necessary checkpoint operations. The checkpointing overhead is a cumulative overhead that takes into account the checkpoint setup, the bus communication, the comparison, and the checkpoint unload

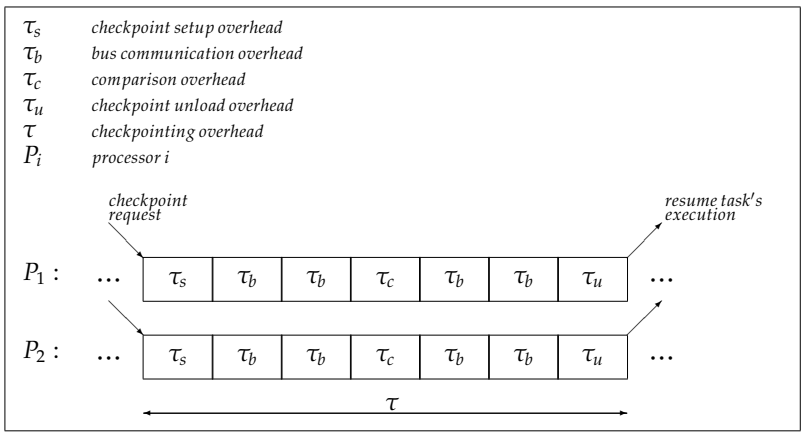


Figure 2.2.: Illustration of checkpointing overhead

overhead. We denote the checkpointing overhead with τ. Figure 2.2 illustrates the checkpointing overhead. As shown in Figure 2.2, at a checkpoint request (the time when a checkpoint is to be taken) both processing nodes spend some time to prepare the checkpoint, *i.e.* checkpoint setup overhead τ_s. Once the checkpoint is ready, one at a time, each processing node transfers the checkpoint over the shared bus, *i.e.* bus communication overhead τ_b. After receiving the checkpoints from both processing nodes, the CCU compares the checkpoints and depending on the comparison, it retrieves the checkpoint that should be sent to both processing nodes, *i.e.* comparison overhead τ_c. The CCU sends the corresponding checkpoint to both processing nodes, one checkpoint at a time for each node, *i.e.* bus communication overhead τ_b. Finally, after both processing nodes have received the checkpoint from the CCU, each processing node extracts the information from the checkpoint and loads this information into its registers, *i.e.* checkpoint unload overhead τ_u. The following expression applies to the checkpointing overhead:

$$\tau = \tau_s + 2\tau_b + \tau_c + 2\tau_b + \tau_u = \tau_s + 4\tau_b + \tau_c + \tau_u \tag{2.2}$$

In RRC, the execution of a job is interleaved with the checkpoint operations, *i.e.* checkpointing overhead is added each time a checkpoint request is issued. Hence, the total execution of a job consists of two parts: useful execution, where the job is executed, and redundant execution, where checkpointing operations are performed. We define the term **execution segment** to refer to the portion of job's execution (useful execution) from the moment when a job is resumed (or started) until a checkpoint request is issued. Thus, the execution

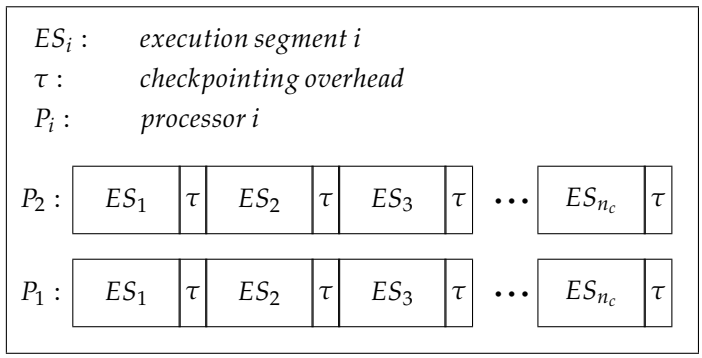


Figure 2.3.: Graphical presentation of RRC scheme

of a job when RRC is employed can be seen as executing a set of execution segments, where each execution segment is followed by a checkpointing overhead. This is illustrated in Figure 2.3. In Figure 2.3, we show the execution of a job when RRC is used with n_c checkpoints. Using RRC with n_c checkpoints results in n_c different execution segments which are denoted with ES_i where $i \in [1, n_c]$. As shown in Figure 2.3, the first execution segment ES_1 is executed on both processing nodes P_1 and P_2. After the execution of ES_1, checkpoints are taken from both processing nodes and the checkpoints are compared against each other. This results in a checkpointing overhead which follows the execution of ES_1 (observe τ follows ES_1 in Figure 2.3). Next, assuming that no errors have occurred, the job proceeds with the execution of the next execution segment ES_2. After the execution of ES_2 another checkpointing overhead τ is added. These steps are repeated until all the execution segments are executed, *i.e.* the job completes after executing ES_{n_c} which is followed by a checkpointing overhead τ. While in Figure 2.3 we showed the execution of a job when RRC is used and no errors occur, in practice, soft errors can occur at any point in time.

Whenever a soft error occurs during the execution of an execution segment, the execution segment is re-executed on both processing nodes. Observe that it is possible that a soft error occurs in only one processing node during the execution of an execution segment. In such scenario, at the end of the execution segment, *i.e.* at a checkpoint request, one of the processing node produces the correct result, while the other processing node produces an erroneous outcome. However, during the comparison performed by the CCU, this error will be detected and re-execution of the execution segment will be enforced on both processing nodes, *i.e.* re-execution of the execution segment will be enforced also on the processing node that has produced the correct outcome.

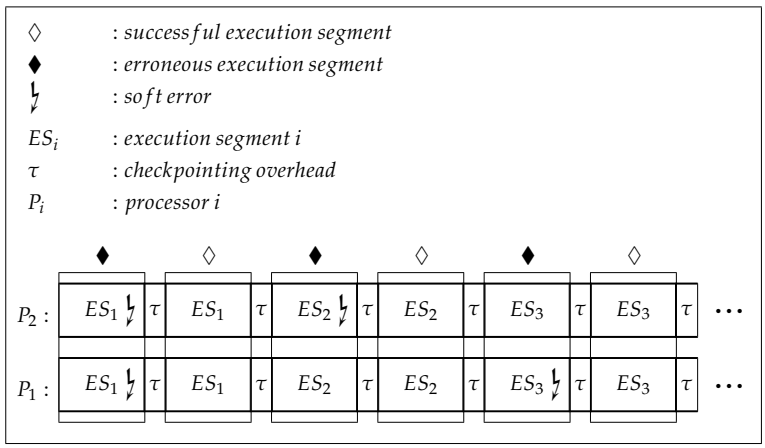


Figure 2.4.: Illustration of successful and erroneous execution segments

In case that soft errors occur in both processing nodes during the execution of an execution segment, following the assumption that each soft error produces a unique erroneous outcome, after the execution of the execution segment, during the comparison, the checkpoints from both processing nodes will differ, meaning that the errors will be detected, and therefore re-execution of the execution segment will be enforced on both processing nodes.

We refer to an execution segment as an **erroneous execution segment** if soft errors have occurred during the execution in at least one of the two processing nodes, and we define a **successful execution segment** as an execution segment where no errors have occurred in any of the processing nodes. Illustration of successful and erroneous execution segments is depicted in Figure 2.4. As shown in Figure 2.4, the first attempt to execute the first execution segment ES_1 results in an erroneous execution segment as errors occur in both processing nodes. Using the assumption that each error results in a unique erroneous outcome, the checkpoints from both processing nodes will be different, and therefore ES_1 has to be re-executed. During the re-execution of ES_1, no errors occur in any of the processing nodes, and this results in a successful execution segment. Since ES_1 is successfully executed, the job proceeds with the next execution segment ES_2. The first attempt to execute ES_2 results in an erroneous execution segment as one error occurs in the processing node P_2, and therefore ES_2 has to be re-executed. During the re-execution of ES_2, no errors occur in any of the processing nodes which results in a successful execution segment. Next, the job proceeds with the execution of the next execution seg-

ment ES_3. The first attempt to execute ES_3 results in an erroneous execution segment as one error occurs in the processing node P_1. In the second attempt to execute ES_3, no errors occur in any of the processing nodes which results in a successful execution segment.

Part I

Equidistant Checkpointing

In this part, we discuss optimization of RRC while assuming that the checkpoints are evenly distributed throughout the execution of the job, *i.e.* equidistant checkpointing. In equidistant checkpointing, all execution segments are of equal length. For equidistant checkpointing, we discuss three optimization objectives for RRC: (1) *Average Execution Time*, (2) *Level of Confidence*, and (3) *Guaranteed Completion Time*. A separate chapter is dedicated to each of the three optimization objectives. For each optimization objective, the problem formulation is provided and a solution is proposed. Each solution is validated with some experimental results. Finally, a summary of the part is provided.

3

Average Execution Time

While RRC is able to cope with soft errors, it introduces a time overhead that impacts the execution time. The time overhead depends on the number of checkpoints that are taken during the execution of a job. In this chapter, we provide a mathematical framework to calculate the optimal number of checkpoints that results in the minimal average execution time (AET). The chapter is organized as follows. We provide the problem formulation in Section 3.1. Derivation of a mathematical expression for calculating the AET of a job when RRC is employed is presented in Section 3.2. In Section 3.3, we provide mathematical formulas for calculating (1) the optimal number of checkpoints and (2) the minimal AET. Finally, experimental results are presented in Section 3.4.

3.1. PROBLEM FORMULATION

In this chapter we discuss the following problem. Given the following inputs:

- a job with a processing time T, *i.e.* the fault-free (error-free) execution time of a job when RRC is not employed,
- a probability P_t the that no soft errors occur in a processing node within an interval of length t,
- a checkpoint setup overhead τ_s,
- a bus communication overhead τ_b,
- a comparison overhead τ_c, and
- a checkpoint unload overhead τ_u,

compute the optimal number of checkpoints n_c that minimizes the AET.

3.2. CALCULATING THE AVERAGE EXECUTION TIME

In this section, we elaborate on how to calculate the AET for a job that employs RRC. The AET depends on multiple parameters among which one important parameter is the number of checkpoints that are taken during the execution of the job. While the AET depends on multiple parameters (for example, the processing time T), in the stated problem formulation we assume that all other parameters, except for the number of checkpoints, are given. We denote the number of checkpoints with n_c, and we assume that equidistant checkpointing is used. In equidistant checkpointing, all execution segments (see definition in Chapter 2) are of the same length. Provided that the processing time T of the job is given, *i.e.* the time needed for a job to complete when no errors occur during the execution and RRC is not employed, and considering that n_c checkpoints are to be used, we calculate the length of a single execution segment t_{ES} as expressed in Eq. (3.1).

$$t_{ES} = \frac{T}{n_c} \tag{3.1}$$

Due to the fact that soft errors can occur during the execution of an execution segment, we need to evaluate the probability of a successful execution segment (see the definition of successful execution segment in Chapter 2). In our problem formulation, given is the probability P_t that no errors occur in a processing node within an interval of length t. Considering the fault model presented in Chapter 2, we first compute the probability P_T that no errors occur in a processing node within an interval of length equal to the processing time of the job T. The probability P_T is computed according to Eq. (3.2).

$$P_T = P_t^{\frac{T}{t}} \tag{3.2}$$

Next, we compute the probability p of an error-free execution segment by using Eq. (3.3). Important to note is that p denotes the probability of an error-free execution segment which is not equivalent to a successful execution segment. While an error-free execution segment refers to an execution segment that has been executed on one processing node and no errors have occurred during the execution, successful execution segment refers to an execution segment that has been successfully (without any errors) executed on both processing nodes. To compute the probability p of an error-free execution segment we use the probability P_T, and p is evaluated as:

$$p = P_T^{\frac{t_{ES}}{T}} = P_T^{\frac{T/n_c}{T}} = P_T^{\frac{1}{n_c}} \tag{3.3}$$

Finally, to compute the probability of a successful execution segment P_ϵ, we

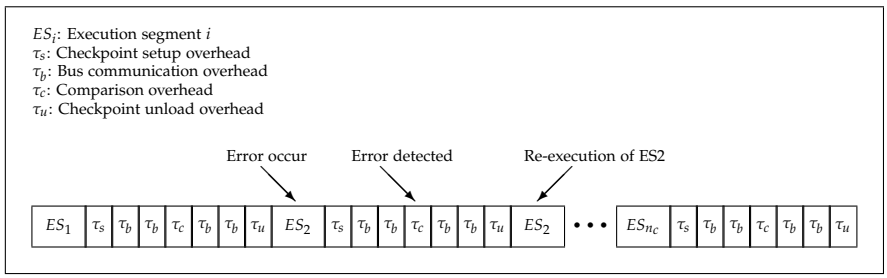


Figure 3.1.: Detailed execution of a job employing RRC with n_c checkpoints

use the expression presented in Eq. (3.4).

$$P_\epsilon = p^2 = {P_T}^{\frac{2}{n_c}} = \sqrt[n_c]{{P_T}^2} \tag{3.4}$$

As shown in Eq. (3.4), the probability of a successful execution segment is calculated as the joint probability that both processing nodes have executed an error-free execution segment, *i.e.* no errors have occurred in any of the processing nodes during the execution of the execution segment. Having the expression for computing P_ϵ, we compute the probability of an erroneous execution segment, denoted with Q_ϵ, by using Eq. (3.5).

$$Q_\epsilon = 1 - P_\epsilon \tag{3.5}$$

Considering that n_c checkpoints are used implies that the job is divided into n_c different execution segments. In such case, the job can complete only after n_c successful execution segments have been executed. However, due to errors, a number of erroneous execution segments might have been executed, and each erroneous execution segment must be re-executed. An illustration of the execution of a job employing RRC, assuming that n_c checkpoints are used, is depicted in Figure 3.1. As illustrated in Figure 3.1, an error has occurred during the execution of the execution segment ES_2, and therefore this execution segment is re-executed.

In general, each of the n_c different execution segments must be executed at least once. However, due to errors, an execution segment may need to be executed several times. Therefore, we first need to compute the expected number of times an execution segment has to be executed. For that reason, we make use of a random variable $\mathbb{X}$ which represents the number of times an execution segment has to be executed. The set of values that can be assigned to the random variable $\mathbb{X}$ is $\mathrm{S} = [1, \infty)$, and this is because an execution

segment must be executed at least once, but may need to be executed infinite number of times. Further, for the random variable $\mathbb{X}$ there exists a probability distribution function $P(\mathbb{X} = k)$ which for each value k in the set S represents the probability that the actual value of the random variable $\mathbb{X}$ is equal to k. Relating this to our particular case, we need to identify the probability distribution function that provides the probability that an execution segment has to be executed a number of times.

We derive the probability distribution function $P(\mathbb{X} = k)$ step by step. In case an execution segment has to be executed once, the only alternative is that the executed execution segment must have been a successful execution segment. Since we have already introduced the probability P_ϵ of a successful execution segment, we evaluate $P(\mathbb{X} = 1)$ as P_ϵ. In case an execution segment has to be executed twice, it means that the execution segment has to be re-executed once since an error has occurred during the first execution. However, no errors have occurred during the re-execution (the second execution). Thus, if an execution segment has been executed twice, the first execution has resulted in an erroneous execution segment, while the second execution has resulted in a successful execution segment. Therefore, $P(\mathbb{X} = 2)$ can be calculated as the probability of having one erroneous and one successful execution segment, *i.e.* $P(\mathbb{X} = 2) = (1 - P_\epsilon) \times P_\epsilon$. In the general case where an execution segment has to be executed k number of times, this means that in the first $k - 1$ executions of the execution segment errors have occurred ($k - 1$ erroneous execution segments have been executed), and only the last execution has been successful (the last execution segment has been a successful execution segment). Hence, we calculate the probability that an execution segment has been executed k number of times as the joint probability of having $k - 1$ erroneous execution segments and a single successful execution segment, and the expression is presented in Eq. (3.6).

$$P(\mathbb{X} = k) = (1 - P_\epsilon)^{k-1} \times P_\epsilon \tag{3.6}$$

Finally, computing the expected number of times an execution segment has to be executed is the same as computing the expected value for the random variable $\mathbb{X}$, which is given in Eq. (3.7).

$$E[\mathbb{X}] = \sum_{k \in S} k \times P(\mathbb{X} = k) \tag{3.7}$$

By replacing $P(\mathbb{X} = x_i)$ (Eq. (3.6)) in Eq. (3.7), we obtain a closed-form expression to compute the expected number of times an execution segment has to be executed, denoted with $E[\mathbb{X}]$, and this expression is presented in Eq. (3.8).

$$
\begin{aligned}
E[\mathbb{X}] &= \sum_{k=1}^{\infty} k \times (\overbrace{1-P_\epsilon}^{Q_\epsilon})^{k-1} \times P_\epsilon = \\
&= P_\epsilon \times \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(\sum_{k=1}^{\infty} Q_\epsilon^k \right) = \\
&= P_\epsilon \times \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(\frac{Q_\epsilon}{1-Q_\epsilon} \right) = \\
&= P_\epsilon \times \frac{1}{1-Q_\epsilon} + P_\epsilon \times \frac{Q_\epsilon}{(1-Q_\epsilon)^2} = \\
&= 1 + \frac{Q_\epsilon}{1-Q_\epsilon} = \frac{1}{1-Q_\epsilon} = \frac{1}{P_\epsilon}
\end{aligned} \tag{3.8}
$$

Let us examine Eq. (3.8) in detail. The first part of Eq. (3.8) is shown in Eq. (3.9). The execution of an execution segment can either be successful, or erroneous. In the case of a successful execution, the execution of the job proceeds with the following execution segment. In the case of an erroneous execution, the execution of the job proceeds by re-executing the erroneous execution segment. The re-execution of the erroneous execution segment can be either successful or erroneous. The re-execution proceeds until the outcome of the latest re-execution has been identified as successful. Eq. (3.9) captures all possible outcomes. In Figure 3.2, we illustrate the possible outcomes when executing an execution segment (ES_i), and we show the expressions, for each possible outcome, that contribute to obtaining $E[\mathbb{X}]$. For $k = 1$, the equation in Figure 3.2 shows the expression for executing the execution segment ES_i only once, while for $k = j$ the equation shows the expression when ES_i is executed j times. As all possible outcomes may occur, a sum over all cases is required, and such a sum results in the expression given in Eq. (3.9).

$$
E[\mathbb{X}] = \sum_{k=1}^{\infty} k \times (\overbrace{1-P_\epsilon}^{Q_\epsilon})^{k-1} \times P_\epsilon \tag{3.9}
$$

Next, we detail the second equilibrium sign in Eq. (3.8). By using the rule to calculate derivative of power functions, we rewrite the expression $k \times Q_\epsilon^{k-1}$ as the derivative of Q_ϵ^k, *i.e.* $\frac{\mathrm{d}}{\mathrm{d}Q_\epsilon}\left(Q_\epsilon^k\right)$, and this is presented in Eq. (3.10).

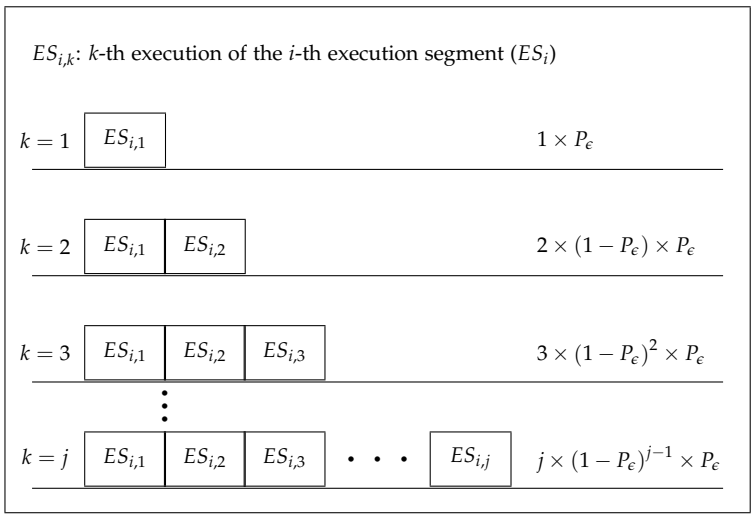


Figure 3.2.: Illustration of possible outcomes when executing an execution segment

$$
\begin{aligned}
& \sum_{k=1}^{\infty} k \times (\overbrace{1 - P_\epsilon}^{Q_\epsilon})^{k-1} \times P_\epsilon = \\
= \quad & P_\epsilon \times \sum_{k=1}^{\infty} k \times Q_\epsilon^{k-1} = \\
= \quad & \left\{ \sum_{k=1}^{\infty} k \times Q_\epsilon^{k-1} = \sum_{k=1}^{\infty} \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(Q_\epsilon^k \right) \right\} = \\
= \quad & P_\epsilon \times \sum_{k=1}^{\infty} \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(Q_\epsilon^k \right)
\end{aligned} \tag{3.10}
$$

By further applying the sum rule in differentiation, we rewrite the expression $\sum_{k=1}^{\infty} \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(Q_\epsilon^k \right)$ in Eq. (3.10) with $\frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(\sum_{k=1}^{\infty} Q_\epsilon^k \right)$, in which case we get the same expression as presented in the second equilibrium sign in Eq. (3.8).

$$
\begin{aligned}
& \sum_{k=1}^{\infty} k \times (\overbrace{1 - P_\epsilon}^{Q_\epsilon})^{k-1} \times P_\epsilon = \\
= \quad & P_\epsilon \times \sum_{k=1}^{\infty} k \times Q_\epsilon^{k-1} =
\end{aligned}
$$

$$
\begin{aligned}
&= \left\{ \sum_{k=1}^{\infty} k \times Q_\epsilon^{k-1} = \sum_{k=1}^{\infty} \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(Q_\epsilon^k \right) \right\} = \\
&= P_\epsilon \times \sum_{k=1}^{\infty} \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(Q_\epsilon^k \right) = \\
&= P_\epsilon \times \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(\sum_{k=1}^{\infty} Q_\epsilon^k \right)
\end{aligned} \tag{3.11}
$$

Eq. (3.12) details the third equilibrium sign in Eq. (3.8). The term $\sum_{k=1}^{\infty} Q_\epsilon^k$ represents a convergent geometric sum, because the common ratio, *i.e.* the ratio between two successive terms from the sum, is lower than one. For the given geometric sum, the common ratio is defined as $\frac{{Q_\epsilon}^{k+1}}{{Q_\epsilon}^k} = Q_\epsilon$. Since Q_ϵ represents the probability of an erroneous execution segment, by definition the following inequality holds $|Q_\epsilon| < 1$. The result of a convergent geometric sum $\sum_{k=0}^{\infty} Q_\epsilon^k$ is equal to $\frac{1}{1-Q_\epsilon}$.

$$
\begin{aligned}
& P_\epsilon \times \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(\sum_{k=1}^{\infty} Q_\epsilon^k \right) = \\
&= P_\epsilon \times \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(Q_\epsilon \times \sum_{k=1}^{\infty} Q_\epsilon^{k-1} \right) = \\
&= P_\epsilon \times \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(Q_\epsilon \times \sum_{k=0}^{\infty} Q_\epsilon^k \right) = \\
&= P_\epsilon \times \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(Q_\epsilon \times \frac{1}{1-Q_\epsilon} \right) = \\
&= P_\epsilon \times \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(\frac{Q_\epsilon}{1-Q_\epsilon} \right)
\end{aligned} \tag{3.12}
$$

Eq. (3.13) details the final part of Eq. (3.8). To obtain the expression in Eq. (3.13) we need to calculate the derivative of the term $\frac{Q_\epsilon}{1-Q_\epsilon}$, which is presented as a product of two functions, *i.e.* $f_1 = Q_\epsilon$ and $f_2 = \frac{1}{1-Q_\epsilon}$. The derivatives of these functions are computed as $\frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} f_1 = 1$ and $\frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} f_2 = \frac{1}{(1-Q_\epsilon)^2}$. We calculate the derivative of the term $\frac{Q_\epsilon}{1-Q_\epsilon}$ by applying the chain rule, and thus we obtain the expression in Eq. (3.13) which provides the same result as presented in Eq. (3.8).

$$
\begin{aligned}
& P_\epsilon \times \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon}\left(\frac{Q_\epsilon}{1-Q_\epsilon}\right) = \\
= \quad & P_\epsilon \times \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon}\left(Q_\epsilon\right) \times \frac{1}{1-Q_\epsilon} + \\
& P_\epsilon \times Q_\epsilon \times \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon}\left(\frac{1}{1-Q_\epsilon}\right) = \\
= \quad & P_\epsilon \times 1 \times \frac{1}{1-Q_\epsilon} + P_\epsilon \times Q_\epsilon \times \frac{1}{(1-Q_\epsilon)^2} = \\
= \quad & P_\epsilon \times \frac{1}{1-Q_\epsilon} + P_\epsilon \times \frac{Q_\epsilon}{(1-Q_\epsilon)^2} = \\
= \quad & \{P_\epsilon = 1-Q_\epsilon\} = \frac{1-Q_\epsilon}{1-Q_\epsilon} + \frac{(1-Q_\epsilon)\times Q_\epsilon}{(1-Q_\epsilon)^2} = \\
= \quad & 1 + \frac{Q_\epsilon}{1-Q_\epsilon} = \frac{1-Q_\epsilon+Q_\epsilon}{1-Q_\epsilon} = \\
= \quad & \frac{1}{1-Q_\epsilon} = \frac{1}{P_\epsilon}
\end{aligned}
\tag{3.13}
$$

The expression that calculates the expected number of times an execution segment has to be executed allows to calculate the expected (average) time that is spent only on execution of execution segments. We denote this expected time as T_{ES}, and we provide the expression that calculates T_{ES} in Eq. (3.14).

$$
\begin{aligned}
T_{ES} \quad = \quad & n_c \times t_{ES} \times E[\mathbb{X}] = \\
= \quad & \frac{n_c \times t_{ES}}{P_\epsilon} = \frac{n_c \times t_{ES}}{p^2} = \\
= \quad & \frac{T}{\sqrt[n_c]{P_T}^2}\ ,\ 0 < P_T < 1
\end{aligned}
\tag{3.14}
$$

Eq. (3.14) takes into account that a total of n_c different execution segments are to be executed, where each of these execution segments has the length t_{ES}, and further each execution segment can be executed several times ($E[\mathbb{X}]$). Important to note is that T_{ES} does not include the time spent on performing checkpointing operations which introduce an extra time overhead. Therefore, next we show how to calculate the average time that is spent on performing the checkpointing operations, denoted with T_{CO}.

In Figure 3.1, we have already shown that after an execution segment is executed, no matter if errors have occurred or not during the execution, a checkpointing overhead is added. The checkpointing overhead includes: the time

to prepare the checkpoints (τ_s, checkpoint setup overhead), the time to transfer the checkpoints over the shared bus (τ_b, bus communication overhead), the time to compare the checkpoints (τ_c, comparison overhead), and the time to load the checkpoints into the registers of the processing nodes (τ_u, checkpoint unload overhead). According to the problem formulation presented in Section 3.1, τ_s, τ_b, τ_c, and τ_u are known in advance, and they all depend on the system's parameters. As these overheads are added after the execution of each execution segment, we calculate T_{CO} by using the expression presented in Eq. (3.15).

$$\begin{aligned} T_{CO} &= n_c \times E[\mathbb{X}] \times (\tau_s + \tau_c + \tau_u + 4 \times \tau_b) = \\ &= \frac{n_c}{P_\epsilon} \times (\tau_s + \tau_c + \tau_u + 4 \times \tau_b) = \\ &= \frac{n_c}{p^2} \times (\tau_s + \tau_c + \tau_u + 4 \times \tau_b) = \\ &= \frac{n_c}{\sqrt[n_c]{P_T}^2} \times (\tau_s + \tau_c + \tau_u + 4 \times \tau_b) \ , 0 < P_T < 1 \end{aligned} \tag{3.15}$$

Finally, the AET for a job employing RRC is the sum of the average time spent only on execution of execution segments T_{ES} and the average time spent only on performing checkpointing operations T_{CO}. Having this said, we present the expression for computing the AET in Eq. (3.16).

$$AET = \frac{T}{\sqrt[n_c]{P_T}^2} + \frac{n_c}{\sqrt[n_c]{P_T}^2} \times (\tau_s + \tau_c + \tau_u + 4 \times \tau_b) \tag{3.16}$$

Given the expressions for T_{ES}, T_{CO}, and AET in Eq. (3.14), Eq. (3.15) and Eq. (3.16), respectively, for the following set of inputs:

- $T = 500$ time units (t.u.), processing time of a job
- $P_t = 0.9$, probability that no errors occur in a processing node within an interval of length 100 t.u.
- $\tau_s = 3$ t.u., checkpoint setup overhead
- $\tau_b = 3$ t.u., bus communication overhead
- $\tau_c = 3$ t.u., comparison overhead
- $\tau_u = 2$ t.u., checkpoint unload overhead

we plot the graphs for T_{ES}, T_{CO}, and AET as functions of n_c.

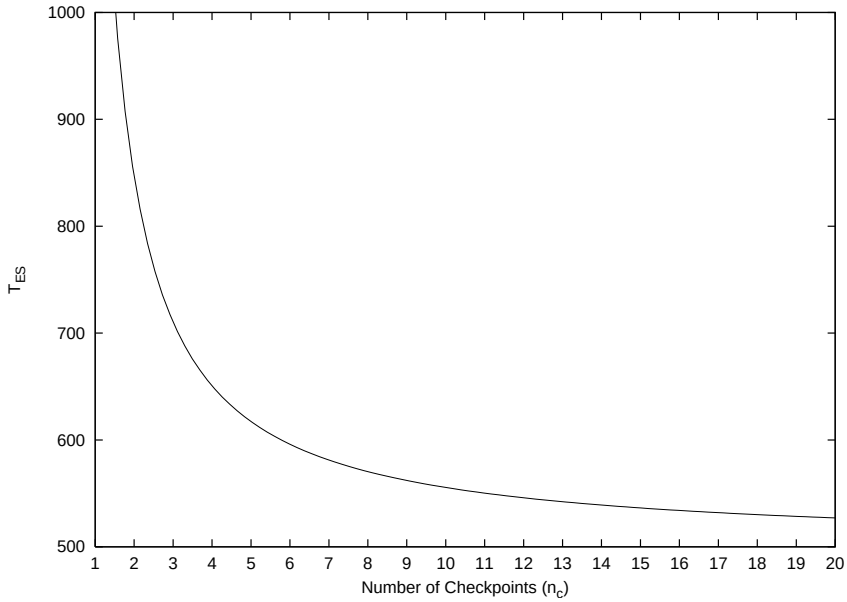


Figure 3.3.: T_{ES}, average time that is spent only on execution of execution segments

The plots for T_{ES}, T_{CO}, and AET are presented in Figure 3.3, Figure 3.4 and Figure 3.5, respectively. In Figure 3.3, the horizontal axis represents the number of checkpoints n_c, while the vertical axis represents the average time spent only on execution of execution segments T_{ES}. As illustrated in Figure 3.3, T_{ES} decreases as the number of checkpoints increases. The reason for this comes from the fact that when many checkpoints are used, the execution segments become shorter. For shorter execution segments the probability of a successful execution segment P_e (see Eq. (3.4)) increases (converges to one) and due to this, the expected number of times that an execution segment has to be executed $E[\mathbb{X}]$, which is the inverse of P_e (see Eq. (3.8)) decreases (converges also to one). Since T_{ES} is computed as the product of the processing time T and $E[\mathbb{X}]$, T_{ES} decreases (converges to T) when $E[\mathbb{X}]$ decreases (converges to one), *i.e.* when n_c is larger. However, for lower values of n_c, the execution segments become larger and therefore, P_e decreases. Since $E[\mathbb{X}]$ is inversely proportional to P_e, $E[\mathbb{X}]$ increases as P_e decreases. As $E[\mathbb{X}]$ increases, T_{ES} also increases due to the fact that T_{ES} is proportional to $E[\mathbb{X}]$. Thus, to reduce T_{ES} it is preferable to use a high number of checkpoints (see the decreasing trend of T_{ES} with respect to n_c in Figure 3.3).

Figure 3.4 shows the average time spent only on performing the check-

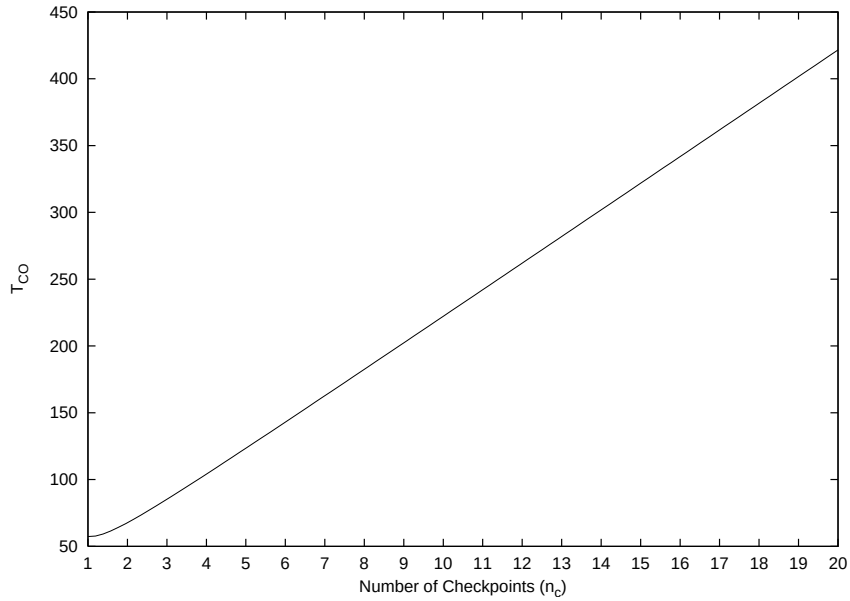


Figure 3.4.: T_{CO}, average time that is spent only on performing checkpointing operations

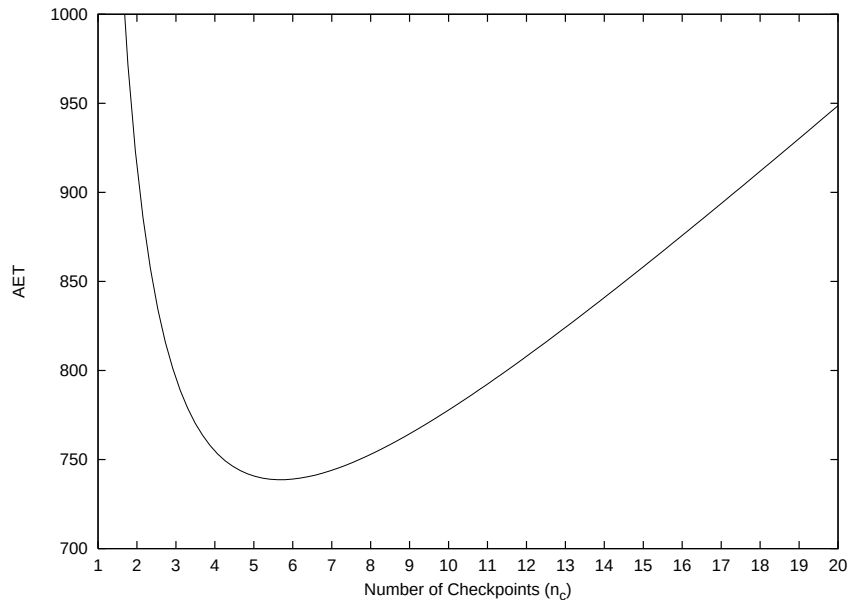


Figure 3.5.: AET, average execution time of a job employing RRC

pointing operations T_{CO} as a function of the number of checkpoints n_c. The horizontal axis represents the number of checkpoints n_c, while the vertical axis represents T_{CO}. As illustrated in Figure 3.4, T_{CO} increases with n_c. The reason that T_{CO} increases with n_c is due to that taking more checkpoints increases the time overhead. Therefore, to reduce T_{CO} it is preferable to use a low number of checkpoints (see the increasing trend of T_{CO} with respect to n_c in Figure 3.4).

In Figure 3.5, we show the AET (the vertical axis) as a function of the number of checkpoints n_c (the horizontal axis). As shown in Figure 3.5, the AET decreases as n_c increases up to a certain breaking point. However, increasing n_c beyond the breaking point increases the AET. This clarifies that there exists an optimal number of checkpoints that minimizes the AET. Obtaining the expressions for calculating the optimal number of checkpoints and the minimal AET is discussed in the following section.

3.3. THE OPTIMAL NUMBER OF CHECKPOINTS

Before we proceed with the steps required to obtain the optimal number of checkpoints, let us first discuss the reason why such optimal number of checkpoints exists. For this, let us first focus on the time overhead. Reducing the time overhead reduces the AET, which is our goal.

The time overhead can be divided into two parts: one part is the overhead caused by performing the checkpointing operations, and the other part is the overhead caused by re-executing the erroneous execution segments. Both parts are tightly related to the number of checkpoints. When the number of checkpoints is high, the overhead due to checkpointing increases because the checkpointing operations are performed more frequently. At the same time, when the number of checkpoints is high, the overhead due to re-execution of erroneous execution segments is reduced because the execution segments are shorter (expressed in Eq. (3.1)). On the other hand, when the number of checkpoints is low, the overhead due to checkpointing decreases because the checkpointing operations are performed less frequently. However, when the number of checkpoints is low, the overhead due to re-execution of erroneous execution segments increases because the execution segments are longer.

We explain this with an example. Assume a job with a processing time $T = 1000$ t.u., a checkpointing overhead $\tau = 50$ t.u., and assume that we need to select between these two alternatives: (1) use a single checkpoint ($n_c = 1$), and (2) use five checkpoints ($n_c = 5$). For the first alternative ($n_c = 1$), the length of the execution segment (only one segment for this alternative) is the same as the processing time of the job, *i.e.* $t_{ES} = 1000$ t.u., while for the second alternative ($n_c = 5$), the length of the execution segments (five

execution segments for this alternative) is $t_{ES} = 200$ t.u. For both alternatives, we compare the completion time, *i.e.* the time needed for the job to complete, for the following two scenarios: (1) no errors occur during the execution of the job, and (2) a single error occurs during the execution of the job.

For the first scenario, where no errors occur during the execution of the job, the completion time is computed as follows:

- for the first alternative ($n_c = 1$), the job completes after executing a single execution segment, followed by a single checkpointing overhead. The completion time is evaluated as $1000 \times 1 + 50 \times 1 = 1050$ t.u.

- for the second alternative ($n_c = 5$), the job completes after executing five execution segments each followed by a checkpointing overhead, and hence the completion time is evaluated as $200 \times 5 + 50 \times 5 = 1250$ t.u.

Important to note is that for this scenario, the only overhead that is added is the overhead caused by performing the checkpointing operations. Since in the second alternative the checkpointing operations are performed five times ($n_c = 5$), the completion time for this alternative is higher when compared against the completion time for the other alternative ($n_c = 1$), where the checkpointing operations are performed only once. From this scenario, we draw the conclusion that an alternative with a lower number of checkpoints provides lower time overhead, and therefore lower completion time.

For the second scenario, where a single error occurs during the execution of the job, the completion time is computed as follows:

- for the first alternative ($n_c = 1$), the single execution segment must be re-executed since an error has occurred during the execution, and therefore the completion time is evaluated as $1000 \times 2 + 50 \times 2 = 2100$ t.u.

- for the second alternative ($n_c = 5$), only one execution segment must be re-executed, and therefore six execution segments are executed in total. The completion time is evaluated as $200 \times 6 + 50 \times 6 = 1500$ t.u.

For this scenario, the time overhead consists of two parts: (1) the overhead caused by performing the checkpointing operations, and (2) the overhead caused by re-executing the erroneous execution segments. For the first alternative ($n_c = 1$), the time overhead is 1100 t.u. where the overhead of performing the checkpointing operations contributes with $50 \times 2 = 100$ t.u., and the overhead due to re-execution of the single erroneous execution segment contributes with 1000 t.u. For the second alternative ($n_c = 5$), the time overhead is 500 t.u. where the overhead of performing the checkpointing operations contributes with $50 \times 6 = 300$ t.u., and the overhead due to re-execution of the erroneous execution segment contributes with 200 t.u. The

reason that the completion time for the second alternative ($n_c = 5$) is lower than the completion time obtained for the first alternative ($n_c = 1$) is because the time overhead due to re-execution of erroneous execution segments is reduced when the number of checkpoints is higher. Hence, from this scenario, we conclude that an alternative with a higher number of checkpoints can considerably reduce the time overhead, and therefore the completion time can be reduced (the completion time when $n_c = 5$ is 1250 t.u., while the completion time when $n_c = 1$ is 2100 t.u.).

The previously discussed example points out that there is a trade-off that is related to the overhead caused by performing the checkpointing operations, and the overhead caused due to re-execution of erroneous execution segments. To clarify this trade-off, we examine Figure 3.3 and Figure 3.4. As can be seen from Figure 3.3, increasing the number of checkpoints *reduces* the average time that is spent only on execution of execution segments T_{ES} and this is due to the fact that the length of the execution segments becomes shorter when the number of checkpoints is higher. Observe that T_{ES} includes also the time that is spent on re-execution of erroneous execution segments, which is part of the time overhead. On the other hand, one can observe from Figure 3.4, that increasing the number of checkpoints *increases* the average time that is spent only on performing checkpointing operations T_{CO}, and this is due to the fact that the checkpointing operations will be performed more frequently when the number of checkpoints is larger. This trade-off motivates the existence of an optimal number of checkpoints that minimizes the AET, and next we show how to compute the optimal number of checkpoints and the minimal AET.

By closely observing the expression for the AET, given in Eq. (3.16), one notes that AET depends on multiple parameters including the number of checkpoints n_c. Since in our problem formulation we consider that all the other parameters, except n_c, are given, we can consider that AET is a single-variable function of n_c. To find the optimum (minimum) of this function we need to compute the first derivative with respect to n_c and set it to be equal to zero.

The first derivative of AET is provided in Eq. (3.17).

$$\begin{aligned}
&\frac{\mathrm{d}AET}{\mathrm{d}n_c} = \\
= \quad & \frac{\mathrm{d}}{\mathrm{d}n_c}\left(\frac{T}{\sqrt[n_c]{P_T}^2} + \overbrace{(\tau_s + \tau_c + \tau_u + 4 \times \tau_b)}^{\tau} \times \frac{n_c}{\sqrt[n_c]{P_T}^2}\right) = \\
= \quad & \tau \times {P_T}^{-2/n_c} + (T + \tau \times n_c) \times \frac{\mathrm{d}}{\mathrm{d}n_c}\left({P_T}^{-\frac{2}{n_c}}\right) = \\
= \quad & \left\{x = -\frac{2}{n_c} \Rightarrow \frac{\mathrm{d}x}{\mathrm{d}n_c} = \frac{2}{n_c^2} \Leftrightarrow \mathrm{d}n_c = \frac{n_c^2}{2}\mathrm{d}x\right\} =
\end{aligned}$$

$$
\begin{aligned}
&= \tau \times {P_T}^{-\frac{2}{n_c}} + (T + \tau \times n_c) \times \frac{2}{n_c^2} \times \frac{\mathrm{d}}{\mathrm{d}x}\left({P_T}^x\right) = \\
&= \tau \times {P_T}^{-\frac{2}{n_c}} + (T + \tau \times n_c) \times \frac{2}{n_c^2} \times (\ln P_T) \times {P_T}^x = \\
&= \tau \times {P_T}^{-\frac{2}{n_c}} + (T + \tau \times n_c) \times \frac{2}{n_c^2} \times (\ln P_T) \times {P_T}^{-\frac{2}{n_c}} = \\
&= {P_T}^{-\frac{2}{n_c}} \times \left(\tau + (T + \tau \times n_c) \times \frac{2}{n_c^2} \times (\ln P_T)\right) \qquad (3.17)
\end{aligned}
$$

To obtain the number of checkpoints that finds the optimum of AET, we need to find the roots of the equation presented in Eq. (3.17). This is given in Eq. (3.18)

$$
\begin{aligned}
0 &= {P_T}^{-\frac{2}{n_c}}(\tau + (T + \tau \times n_c) \times \frac{2}{n_c^2} \times (\ln P_T)) = \\
&= \tau + (T + \tau \times n_c) \times \frac{2}{n_c^2} \times (\ln P_T) = \\
&= \frac{\tau}{(\ln P_T)} + (T + \tau \times n_c) \times \frac{2}{n_c^2} = \\
&= \frac{\tau \times n_c^2}{(\ln P_T)} + 2 \times T + 2 \times \tau \times n_c = \\
&= n_c^2 + 2 \times (\ln P_T) \times n_c + \frac{2 \times T \times (\ln P_T)}{\tau} = \\
&= (n_c + (\ln P_T))^2 - (\ln P_T)^2 + \frac{2 \times T \times (\ln P_T)}{\tau} \Leftrightarrow \\
n_c^s &= -(\ln P_T) + \sqrt{(\ln P_T)^2 - \frac{2 \times T \times (\ln P_T)}{\tau}} = \\
&= -(\ln P_T) + \sqrt{(\ln P_T)^2 - \frac{2 \times T \times (\ln P_T)}{\tau_s + \tau_c + \tau_u + 4 \times \tau_b}} \qquad (3.18)
\end{aligned}
$$

The expression provided in Eq. (3.18) evaluates the stationary point n_c^s for which the function AET reaches its optimum. To ensure that AET reaches its minimum in the stationary point n_c^s, it is necessary to examine the second derivative of AET. If the second derivative, evaluated at the stationary point n_c^s, is positive, then the AET reaches its minimum at n_c^s.

The second derivative of AET is presented in Eq. (3.19). As can be seen from Eq. (3.19), whenever n_c is positive the second derivative is always evaluated as positive. To justify the former statement, let us closely inspect the expression given in Eq. (3.19). The expression presented in Eq. (3.19) represents a product of two terms. The first multiplicand is always negative

because it involves multiplication of the term $\ln P_T$, which is negative due to the fact that $0 < P_T < 1$. The second multiplicand of Eq. (3.19), *i.e.* the term in brackets, consists of three terms, where two of the terms are added together and the third term is subtracted. The terms that are added together are obtained after a multiplication with $(\ln P_T)$, thus each one of them is negative, hence the sum of these two terms is also negative. The third term, *i.e.* $n_c \times T$ is positive, whenever n_c is positive. Subtracting a positive term, *i.e.* $n_c \times T$, from a negative term, *i.e.* the sum of the other two terms, results in a negative term, and therefore the second multiplicand in Eq. (3.19) is negative (under the condition that n_c is positive). Finally, we conclude that the second derivative of AET is evaluated as positive because it is obtained as a product of two negative terms, under the condition that n_c is positive. Since the stationary point n_c^s is evaluated as a positive value, evaluating the second derivative of AET at the stationary point will provide a positive value, which means that the function AET reaches its minimum in the stationary point.

$$\frac{\mathrm{d}^2 AET}{\mathrm{d}n_c^2} = \frac{4 \times {P_T}^{-\frac{2}{n_c}} \times (\ln P_T)}{n_c^4} \times (T \times (\ln P_T) + \tau \times n_c \times (\ln P_T) - n_c \times T) \tag{3.19}$$

Out of this discussion we conclude that the optimal number of checkpoints that results in the minimal AET can be obtained by using the expression given in Eq. (3.18). For clarity, we denote with n_c^* the optimal number of checkpoints that minimizes the AET, and we re-write Eq. (3.18) as follows:

$$n_c^* = -(\ln P_T) + \sqrt{(\ln P_T)^2 - \frac{2 \times T \times (\ln P_T)}{\tau}} \tag{3.20}$$

Finally, the expression to calculate the optimal number of checkpoints n_c^* (Eq. (3.20)) allows us to calculate the minimal AET. We denote the minimal AET with AET^* and compute it according to the expression presented in Eq. (3.21).

$$AET^* = \frac{T}{\sqrt[n_c^*]{P_T}^2} + \frac{n_c^*}{\sqrt[n_c^*]{P_T}^2} \times (\tau_s + \tau_c + \tau_u + 4 \times \tau_b) \tag{3.21}$$

Observe that according to Eq. (3.20) n_c^* may be evaluated as a real number. However, if there is a practical limitation that the optimal number of checkpoints should be an integer, then we need to examine AET for the lower and the upper integer bound of n_c^*. This is needed because rounding n_c^* to the closest integer number does not always result in the minimal AET.

Scenario	T	τ	P_T
A	1000 t.u.	100 t.u.	0.9
B	1000 t.u.	10 t.u.	0.9
C	1000 t.u.	100 t.u.	0.7
D	1000 t.u.	10 t.u.	0.7

Table 3.1.: Input scenarios

3.4. EXPERIMENTAL RESULTS

The purpose of the results presented in this section is to show that the AET varies with the number of checkpoints, and to show that the optimal number of checkpoints that results in the minimal AET can be obtained by using the expression provided in Eq. (3.20). For that reason, we present results for the input scenarios summarized in Table 3.1. For each input scenario, the following three parameters are given: the processing time of the job T, the checkpointing overhead τ and the error-free probability P_T that no errors occur in a processing node within an interval of length T.

The input scenarios in Table 3.1 were chosen such that the impact of different error-free probabilities and different ratios $\frac{T}{\tau}$ on the AET can be explored. For that reason, we have considered two different error-free probabilities, namely $P_T = 0.9$ used in Scenario A and B, and $P_T = 0.7$ used in Scenario C and D, and two different ratios $\frac{T}{\tau}$, namely $\frac{T}{\tau} = 10$ used in Scenario A and C, and $\frac{T}{\tau} = 100$ used in Scenario B and D.

For the given input scenarios, we calculate the AET, using the expression in Eq. (3.16), for different number of checkpoints. Tables 3.2–3.5 show the AET obtained for $n_c \in [1, 20]$ for the given input scenarios A–D, respectively. As can be seen in Tables 3.2–3.5, the AET varies with the number of checkpoints, and there exists an optimal number of checkpoints that minimizes the AET.

Using Eq. (3.18), we compute the optimal number of checkpoints that results in the minimal AET for each of the presented scenarios. In such case, the following results are obtained:

- $n_c^* = 1.561$ for Scenario A,
- $n_c^* = 4.697$ for Scenario B,
- $n_c^* = 3.051$ for Scenario C, and
- $n_c^* = 8.810$ for Scenario D.

If only integer values are allowed for the number of checkpoints, then we need to evaluate the AET for both the lower and the upper integer bound of

n_c	AET	n_c	AET
1	1358.025	11	2140.616
2	**1333.333**	12	2238.973
3	1394.596	13	2337.585
4	1475.730	14	2436.397
5	1564.567	15	2535.368
6	1657.191	16	2634.469
7	1751.953	17	2733.676
8	1848.042	18	2832.971
9	1945.010	19	2932.342
10	2042.591	20	3031.775

Table 3.2.: AET obtained for $n_c \in [1, 20]$, for Scenario A

n_c	AET	n_c	AET
1	1246.914	11	1131.469
2	1133.333	12	1139.841
3	1104.945	13	1148.466
4	1096.256	14	1157.288
5	**1095.197**	15	1166.269
6	1097.889	16	1175.378
7	1102.700	17	1184.593
8	1108.825	18	1193.895
9	1115.822	19	1203.271
10	1123.425	20	1212.710

Table 3.3.: AET obtained for $n_c \in [1, 20]$, for Scenario B

n_c^*, *i.e.* ($\lfloor n_c^* \rfloor$) and $\lceil n_c^* \rceil$, and report the integer bound of n_c^* that provides the lower AET. For example, in Scenario A, n_c^* is evaluated as 1.561, and therefore we need to evaluate the AET for $n_c = 1$ and $n_c = 2$. Since for these two values $n_c = 1$ and $n_c = 2$ lower AET is obtained for $n_c = 2$, the upper bound of n_c^* will be used. The optimal number of checkpoints for each of the given scenarios is as follows:

- $n_c = 2$ for Scenario A,
- $n_c = 5$ for Scenario B,
- $n_c = 3$ for Scenario C, and
- $n_c = 9$ for Scenario D.

n_c	AET	n_c	AET
1	2244.898	11	2240.698
2	1714.286	12	2334.746
3	**1648.965**	13	2429.735
4	1673.320	14	2525.458
5	1730.024	15	2621.764
6	1801.997	16	2718.542
7	1882.377	17	2815.707
8	1967.877	18	2913.194
9	2056.725	19	3010.949
10	2147.882	20	3108.934

Table 3.4.: AET obtained for $n_c \in [1, 20]$, for Scenario C

n_c	AET	n_c	AET
1	2061.224	11	1184.369
2	1457.143	12	1188.598
3	1306.487	13	1193.739
4	1243.038	14	1199.592
5	1211.017	15	1206.011
6	1193.823	16	1212.888
7	1184.790	17	1220.140
8	1180.726	18	1227.703
9	**1179.911**	19	1235.528
10	1181.335	20	1243.573

Table 3.5.: AET obtained for $n_c \in [1, 20]$, for Scenario D

4

Level of Confidence

In the previous chapter, we presented an optimization approach for RRC with the goal to minimize the AET. In general, minimizing the AET is important as it can improve the system's performance. However, the main drawback with the minimal AET is that we lack the distribution of the execution times, *i.e.* sometimes a job may complete much earlier in time, but sometimes it may take much longer time. Having this said, the minimal AET does not provide any guarantees that time constraints (deadlines) are always met. Even when the AET is lower than the deadline, there are no guarantees that the deadline will not be violated as sometimes a job may take much longer time than the AET. For soft RTSs eventual violation of deadlines does not result in catastrophic outcomes, and it is therefore justified to optimize RRC with the goal to minimize the AET. However, for hard RTSs, violating the deadlines may have severe consequences, and thus optimizing RRC with the goal to minimize the AET is not sufficient. Instead, it is for hard RTSs more important to ensure, with a very high likelihood, that the deadlines are met. To evaluate to what extent a deadline is met, in this section, we use Level of Confidence (LoC) as a metric to evaluate the probability that a given deadline is met.

The time overhead introduced by RRC may be the reason that deadlines in RTSs are violated. Since for hard RTSs it is crucial to meet the deadlines, in this chapter, we optimize RRC with the goal to maximize the LoC. Obtaining the maximal LoC ensures, with the highest probability (often very close to one), that deadlines in hard RTSs are met. The chapter is divided into two sections. First, we discuss the LoC for a single real-time job, *i.e.* the completion of the job is constrained by a given deadline. Second, we discuss the LoC for multiple jobs with respect to a given global deadline.

4.1. SINGLE JOB

In this section, we provide analysis for the LoC for a single real-time job which uses RRC. The section is organized as follows. The problem formulation is stated in Section 4.1.1. In Section 4.1.2, we present a mathematical framework for evaluation of LoC with respect to a given deadline for a single job. Important properties of the expression used to calculate the LoC are discussed in Section 4.1.3. Section 4.1.4 focuses on maximizing the LoC for a single real-time job. Finally, in Section 4.1.5 we show some experimental results.

4.1.1. PROBLEM FORMULATION

In this section we discuss the following problem. Given the following inputs:

- a job with a processing time T,
- a deadline D,
- a checkpointing overhead τ, and
- a probability P_T that no soft errors occur in a processing node within an interval of length T,

compute the optimal number of checkpoints such that the maximal LoC of meeting the given deadline is obtained.

4.1.2. EVALUATION OF THE LEVEL OF CONFIDENCE

In this section, we analyze the LoC for a single real-time job which uses RRC, and we derive an expression to evaluate the LoC with respect to a given deadline D. As the LoC represents the probability that a job completes *before* the deadline, it is determined as a sum of intermediate terms which represent the probability that a job completes *exactly at* a given discrete point in time. These terms are calculated according to a probability distribution function, and this function must satisfy some necessary conditions. Thus, to compute the LoC, we need to derive an expression for the probability distribution function.

This section is organized as follows. First, we construct an expression for the probability distribution function. Next, we prove that the proposed expression can be used as a valid probability distribution function. At the end of this section, we provide an expression to evaluate the LoC.

To derive the probability distribution function, we start by analyzing the expected time for a job to complete. The expected completion time can be described by a discrete variable due to the fact that an integer number of

execution segments (each followed by a checkpointing overhead) must be executed before a job completes. Assuming that n_c checkpoints are to be taken, a job can complete only when n_c successful execution segments have been executed. Thus, in the best case, when no errors occur, a job completes after n_c execution segments have been executed each followed by a checkpointing overhead. In equidistant checkpointing, the size of the execution segments is determined as $\frac{T}{n_c}$. Therefore, the completion time when no errors occur, denoted with ${}^{n_c}t_0$, is calculated with the following expression:

$$ {}^{n_c}t_0 = n_c \times (\frac{T}{n_c} + \tau) = T + n_c \times \tau \tag{4.1} $$

If errors occur, and these errors only affect the execution of *one* execution segment, the affected execution segment is an erroneous execution segment. This execution segment has to be re-executed, and therefore, in total, $n_c + 1$ execution segments will be executed (one erroneous and n_c successful execution segments). We denote the completion time when one execution segment is re-executed, under the assumption that n_c checkpoints are used, with ${}^{n_c}t_1$, and it is calculated as:

$$ {}^{n_c}t_1 = (n_c + 1) \times (\frac{T}{n_c} + \tau) = T + n_c \times \tau + (\frac{T}{n_c} + \tau) \tag{4.2} $$

In general, when there are k erroneous execution segments, ${}^{n_c}t_k$ denotes the expected completion time, and it is calculated as:

$$ {}^{n_c}t_k = T + n_c \times \tau + k \times (\frac{T}{n_c} + \tau) \tag{4.3} $$

Next, we analyze the number of possible cases that a job completes exactly at time ${}^{n_c}t_k$. First, let us study the case that a job completes at time ${}^{n_c}t_0$. This can happen if and only if all the execution segments have been successful, meaning that no errors have occurred during the execution of each of the execution segments. This is the only possible alternative for a job to complete at time ${}^{n_c}t_0$. Now, let us assume that a job completes at time ${}^{n_c}t_1$. If a job completes at time ${}^{n_c}t_1$, a single execution segment has been re-executed. This can be any of the n_c different execution segments. Thus, there are n_c possible cases where a job completes at time ${}^{n_c}t_1$. If a job completes at time ${}^{n_c}t_2$, it means that two execution segments have been re-executed. It can either be that two out of all n_c different execution segments have been re-executed, or a single execution segment has been re-executed twice (an error has been detected after the first re-execution). In general, if a job completes at time ${}^{n_c}t_k$, a total of $n_c + k$ execution segments have been executed, *i.e.* n_c successful execution segments and k erroneous execution segments. Note that the last execution segment among the $n_c + k$ execution segments must have been a

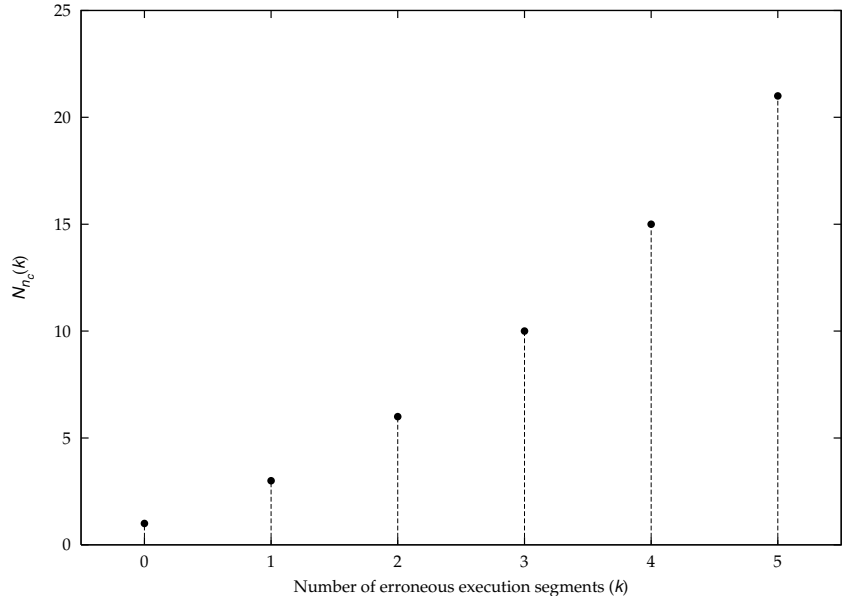


Figure 4.1.: Number of cases $N_{n_c}(k)$ for $n_c = 3$ and $P_T = 0.5$

successful execution segment. Otherwise, if the last execution segment has been erroneous, it contradicts the assumption that the job has completed at ${}^{n_c}t_k$ (an erroneous execution segment always requires a re-execution). Hence, the k erroneous execution segments are any of the $n_c + k - 1$ execution segments (any execution segment except for the last one). Therefore, the number of possible cases such that a job completes at time ${}^{n_c}t_k$ is the number of all the combinations of k execution segments out of $n_c + k - 1$ execution segments. $N_{n_c}(k)$ denotes the number of possible cases that a job completes at time ${}^{n_c}t_k$, and $N_{n_c}(k)$ is defined as

$$N_{n_c}(k) = \binom{n_c + k - 1}{k} \tag{4.4}$$

In Figure 4.1, we illustrate $N_{n_c}(k)$ (Eq. (4.4)) for $n_c = 3$, $P_T = 0.5$, and $k \in [0, 5]$. For example, $N_3(1) = 3$ shows that there are three cases that a job completes at ${}^3t_1$ (one erroneous execution segment has been executed before the job has completed), since any of the three execution segments ($n_c = 3$) could have been re-executed.

Next, to calculate the probability that a job completes at time ${}^{n_c}t_k$, we assign a probability metric for each case where a job completes at ${}^{n_c}t_k$. This probability metric is related to the probability of a successful execution segment P_ϵ.

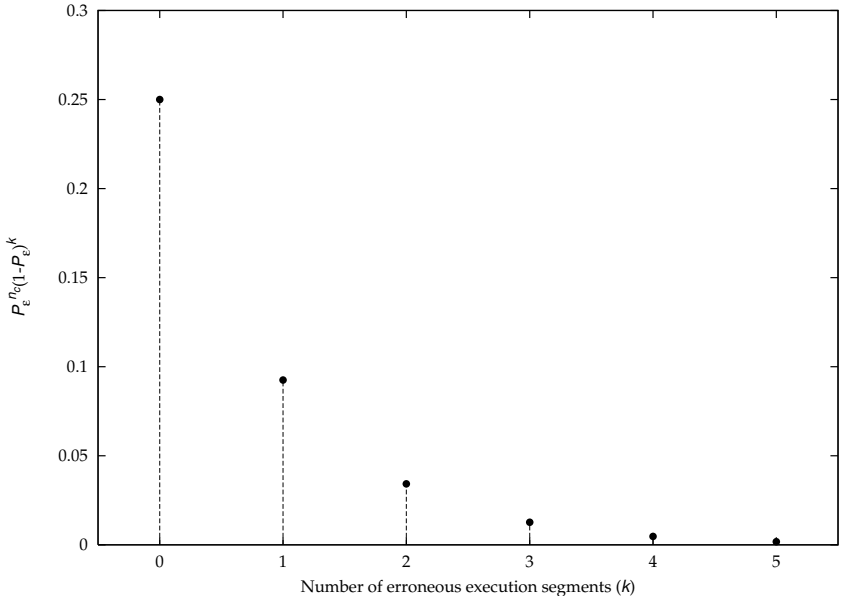


Figure 4.2.: Probability metric per case $P_\epsilon^{n_c} \times (1 - P_\epsilon)^k$ for $n_c = 3$ and $P_T = 0.5$

As discussed in Chapter 3, P_ϵ is derived from the given probability P_T that no errors occur in a processing node within an interval of length T, and P_ϵ is evaluated with the following expression:

$$P_\epsilon = {P_T}^{\frac{2}{n_c}} = \sqrt[n_c]{{P_T}^2} \tag{4.5}$$

When a job completes at time ${}^{n_c}t_k$, $n_c + k$ execution segments are executed, *i.e.* n_c *successful* execution segments and k *erroneous* execution segments. Since P_ϵ represents the probability of a successful execution segment, the probability of an erroneous execution segment is evaluated as $(1 - P_\epsilon)$. Due to the fact that the execution segments are independent, the probability of having n_c successful execution segments is $P_\epsilon^{n_c}$, and the probability of having k erroneous execution segments is $(1 - P_\epsilon)^k$. Combining these two probabilities, the probability of having n_c successful and k erroneous execution segments results in $P_\epsilon^{n_c} \times (1 - P_\epsilon)^k$, which is the probability metric per possible case where a job completes at time ${}^{n_c}t_k$. In Figure 4.2, we illustrate the probability metric per possible case $P_\epsilon^{n_c} \times (1 - P_\epsilon)^k$ for $n_c = 3$, $P_T = 0.5$, and $k \in [0, 5]$. From Figure 4.2 it can be observed that $P_\epsilon^{n_c} \times (1 - P_\epsilon)^k$ has the highest value at $k = 0$, and it is evaluated as $P_\epsilon^{n_c} = \left(\sqrt[n_c]{{P_T}^2}\right)^{n_c} = {P_T}^2 = 0.25$.

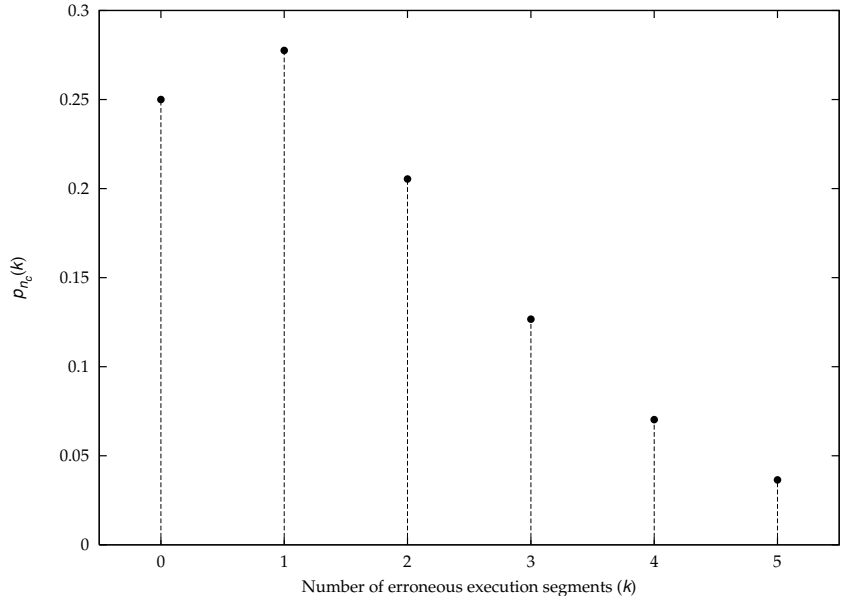


Figure 4.3.: Probability distribution function $p_{n_c}(k)$ for $n_c = 3$ and $P_T = 0.5$

To calculate the probability that a job completes at time ${}^{n_c}t_k$, we need to multiply the number of possible cases $N_{n_c}(k)$ with the probability metric per case $P_\epsilon^{n_c} \times (1 - P_\epsilon)^k$. We denote the probability that a job completes at time ${}^{n_c}t_k$ with $p_{n_c}(k)$, and we define it as:

$$p_{n_c}(k) = N_{n_c}(k) \times P_\epsilon^{n_c} \times (1 - P_\epsilon)^k = \binom{n_c + k - 1}{k} \times P_\epsilon^{n_c} \times (1 - P_\epsilon)^k \quad (4.6)$$

Eq. (4.6) defines the probability distribution function. In Figure 4.3, we illustrate the probability distribution function $p_{n_c}(k)$ for $n_c = 3$, $P_T = 0.5$, and $k \in [0, 5]$.

To verify that the expression in Eq. (4.6) can be used as a valid probability distribution function, we prove that this function satisfies the following necessary condition which a probability distribution function must satisfy:

$$\sum_{k=0}^{\infty} p_{n_c}(k) = 1 \quad (4.7)$$

The proof is provided in Appendix A.

Since Eq. (4.6) represents a valid probability distribution function, to compute the LoC it is required to sum up all the terms from the probability distribution function $p_{n_c}(k)$ (Eq. (4.6)) for which the discrete variable ${}^{n_c}t_k$ has a value which is lower or equal to the given deadline D. We denote with $\Lambda_{n_c}(D)$ the LoC with respect to a deadline D, and we compute $\Lambda_{n_c}(D)$ as:

$$\Lambda_{n_c}(D) = \sum_{{}^{n_c}t_k \leq D} p_{n_c}(k) = \sum_{{}^{n_c}t_k \leq D} \binom{n_c + k - 1}{k} \times P_\epsilon^{n_c} \times (1 - P_\epsilon)^k \tag{4.8}$$

Observe that the given deadline D is not a discrete variable, and therefore it may not necessarily be an instance of the completion time ${}^{n_c}t_k$. On the other hand, when RRC is used, a job can only complete at discrete time instances which are defined with the completion time ${}^{n_c}t_k$.

As expressed in Eq. (4.8), the LoC with respect to a given deadline D is calculated as a sum of terms from the probability distribution function $p_{n_c}(k)$ where each term represents the probability that a job completes at a discrete time ${}^{n_c}t_k$. Further, only the terms from the probability distribution function $p_{n_c}(k)$ for which the discrete time ${}^{n_c}t_k$ is lower or equal to the given deadline D are used to obtain $\Lambda_{n_c}(D)$. For any given n_c, a finite number $k^\dagger$ of instances of the completion time ${}^{n_c}t_k$ satisfy the condition ${}^{n_c}t_k \leq D$, *i.e.* all instances of the completion time ${}^{n_c}t_k$ where $k \leq k^\dagger$ are lower than the deadline D. This implies that $\Lambda_{n_c}(D)$ is calculated as a sum of terms from the probability distribution function $p_{n_c}(k)$ where $k \leq k^\dagger$. Important to note is that for different n_c values it is possible that the same number of instances $k^\dagger$ of the completion time ${}^{n_c}t_k$ satisfy the condition ${}^{n_c}t_k \leq D$. Therefore, for all such n_c values $\Lambda_{n_c}(D)$ will be computed as a sum of $k^\dagger + 1$ terms of $p_{n_c}(k)$, *i.e.*:

$$\Lambda_{n_c}(D) = \sum_{k=0}^{k^\dagger} p_{n_c}(k) \tag{4.9}$$

The LoC with respect to a given deadline D, denoted with $\Lambda_{n_c}(D)$, provides the probability that a job completes before the deadline. On the other hand, the LoC with respect to a given completion time ${}^{n_c}t_{k^\dagger}$ provides that probability that a job completes after at most $k^\dagger$ re-executions (at most $k^\dagger$ erroneous execution segments), and it is calculated as a sum of $k^\dagger + 1$ terms of $p_{n_c}(k)$, *i.e.*:

$$\Lambda_{n_c}({}^{n_c}t_{k^\dagger}) = \sum_{k=0}^{k^\dagger} p_{n_c}(k) \tag{4.10}$$

To distinguish between the LoC with respect to a given deadline D and the LoC with respect to a given completion time ${}^{n_c}t_k$, we introduce another

notation ($\lambda_{n_c}(k)$) when referring to the LoC with respect to a given completion time. We denote with $\lambda_{n_c}(k)$ the LoC with respect to a given completion time ${}^{n_c}t_k$, and we calculate it as:

$$\lambda_{n_c}(k) = \sum_{i=0}^{k} p_{n_c}(i) \tag{4.11}$$

Hence, we use the notation $\lambda_{n_c}(k)$ when we refer to the LoC with respect to a given completion time, and we use the notation $\Lambda_{n_c}(D)$ when we refer to the LoC with respect to a given deadline. Observe that for a given n_c and a given deadline D, the following equality holds:

$$\Lambda_{n_c}(D) = \lambda_{n_c}(k^{\dagger}) \tag{4.12}$$

where $k^{\dagger}$, in Eq. (4.12), represents the maximum number of re-executions, for the given n_c, that can take place without violating the given deadline D.

In the rest of the thesis, we use both notations depending on whether we address the LoC with respect to a given deadline, or the LoC with respect to a given completion time ${}^{n_c}t_k$.

4.1.3. PROPERTIES OF THE LEVEL OF CONFIDENCE

In this section, we present some important properties of the expression which is used to calculate the LoC. We summarize these properties with the following theorems, and we provide a proof for each theorem.

Theorem 1. *For any n_c, the LoC with respect to a completion time (or a deadline) that does not include any re-executions does not vary with n_c and is calculated as:*

$$\lambda_{n_c}(0) = {P_T}^2$$

Proof. Given that the completion time does not include any re-executions implies that $k = 0$. By replacing k with zero and using Eq. (4.5), Eq. (4.6), and Eq. (4.11), we get:

$$\begin{aligned} \lambda_{n_c}(0) &= p_{n_c}(0) \\ &= {P_\epsilon}^{n_c} \\ &= {P_T}^2 \end{aligned} \tag{4.13}$$

As shown in Eq. (4.13), $\lambda_{n_c}(0)$ does not depend on n_c, *i.e.* the right-hand side of Eq. (4.13) is a constant. ■

Theorem 2. *For any n_c, k_1, and k_2, such that $k_1 > k_2$, the following condition holds:*

$$\lambda_{n_c}(k_1) > \lambda_{n_c}(k_2)$$

Proof. The proof comes from the definition of $\lambda_{n_c}(k)$ presented in Eq. (4.11). By using k_1 in Eq. (4.11) and assuming that $k_1 > k_2$, we get:

$$\begin{aligned}\lambda_{n_c}(k_1) &= \sum_{i=0}^{k_1} p_{n_c}(i) \\ &= \sum_{i=0}^{k_2} p_{n_c}(i) + \sum_{i=k_2+1}^{k_1} p_{n_c}(i) \\ &= \lambda_{n_c}(k_2) + \sum_{i=k_2+1}^{k_1} p_{n_c}(i) \end{aligned} \tag{4.14}$$

The right-hand side of Eq. (4.14) is strictly larger than $\lambda_{n_c}(k_2)$ as it obtained as a sum of $\lambda_{n_c}(k_2)$ and a number of terms from the probability distribution function $p_{n_c}(k)$. As each term $p_{n_c}(k)$ represents a probability, it is only allowed to be greater or equal to zero. By this we have shown that for any n_c, k_1, and k_2, such that $k_1 > k_2$, the following relation holds $\lambda_{n_c}(k_1) > \lambda_{n_c}(k_2)$. ■

Theorem 3. *For any k, n_{c_1}, and n_{c_2}, such that $n_{c_1} > n_{c_2}$, the following condition holds:*

$$\lambda_{n_{c_1}}(k) > \lambda_{n_{c_2}}(k)$$

Proof. The proof is presented in Appendix B. ■

Theorem 4. *For any k there exists a limit $\bar{\lambda}_k$, such that for any n_c the following condition holds:*

$$\lambda_{n_c}(k) \leq \bar{\lambda}_k$$

and $\bar{\lambda}_k$ is defined as:

$$\bar{\lambda}_k = {P_T}^2 \sum_{i=0}^{k} \frac{\left(-2\ln(P_T)\right)^i}{i!}$$

Proof. The proof is presented in Appendix C. ■

4.1.4. MAXIMIZING THE LEVEL OF CONFIDENCE

In this section, we provide analysis on how to obtain the optimal number of checkpoints n_c^* that provides the maximal LoC $\Lambda_{n_c^*}(D)$ with respect to a given deadline D. A straightforward approach would require computing $\Lambda_{n_c}(D)$ for all $n_c \geq 1$, *i.e.*

$$\Lambda_{n_c^*}(D) = MAX\left\{\Lambda_{n_c}(D) \mid n_c \geq 1\right\} \tag{4.15}$$

The straightforward approach can be improved by providing an upper bound $n_{c_{max}}$ for the number of checkpoints, and thus it can be expressed as:

$$\Lambda_{n_c^*}(D) = MAX\left\{\Lambda_{n_c}(D) \mid n_c \in [1, n_{c_{max}}]\right\} \tag{4.16}$$

However, even for the improved straightforward approach, the optimal number of checkpoints n_c^* can be obtained after computing $\Lambda_{n_c}(D)$ for $n_{c_{max}}$ different n_c values, *i.e.* $n_c \in [1, n_{c_{max}}]$.

Next, we show that it is possible to obtain the optimal number of checkpoints n_c^* without computing $\Lambda_{n_c}(D)$ for $n_{c_{max}}$ different n_c values. Instead, it is only sufficient to compute $\Lambda_{n_c}(D)$ for a number of specific n_c values. The number of such n_c values is equal to the maximal number K_{max} of re-executions that can take place without violating the given deadline. Important to note is that K_{max} is much lower than $n_{c_{max}}$. Obtaining the optimal number of checkpoints n_c^* by only computing $\Lambda_{n_c}(D)$ for at most K_{max} specific n_c values is possible due to the property of the expression used for evaluation of the LoC, stated with **Theorem 3**.

As shown in Eq. (4.8), $\Lambda_{n_c}(D)$ is defined as a sum of intermediate terms of the probability distribution function (see Eq. (4.6)). As we calculate the LoC with respect to a given deadline, there is a limit on the number of terms that can be included in the sum that calculates the LoC. This limit depends on the number of checkpoints. When RRC is applied with a low number of checkpoints, the execution segments are longer, and thus a single re-execution is more costly in terms of time. This imposes a limit on the number of re-executions that can take place without violating the deadline, and implies that only a few terms from the probability distribution function can be included in the sum. Increasing the number of checkpoints results in shorter execution segments, and therefore more re-executions can take place without violating the deadline which implies that more terms of the probability distribution can be included in the sum. However, increasing the number of checkpoints increases the checkpointing overhead which imposes a limit on the number of re-executions that can take place without violating the deadline. This in turn limits the number of terms from the probability distribution that can be included in the sum. Therefore, we conclude that by increasing the number of checkpoints up to a certain (breaking) point, the number of terms from the probability distribution function included in the sum that calculates the LoC increases along with the number of checkpoints. However, increasing the number of checkpoints beyond the breaking point decreases the number of terms from the probability distribution function that are included in the sum which calculates the LoC. Furthermore, increasing the number of checkpoints excessively may result in a very high checkpointing overhead which results in a deadline violation. From this discussion we draw the following conclusions. First, there exists a maximal number of checkpoints $n_{c_{max}}$ such that for any $n_c > n_{c_{max}}$ the checkpointing overhead violates the deadline. Second, for a given number of checkpoints n_c, where $1 \leq n_c \leq n_{c_{max}}$, there exists a number of re-executions k that can take place without violating the deadline, which leads to the fact that the LoC with respect to the deadline is calculated as a

sum of $k+1$ intermediate terms from the probability distribution function, *i.e.* $\Lambda_{n_c}(D) = \sum_{i=0}^{k} p_{n_c}(i)$.

The maximal number of checkpoints $n_{c_{max}}$ can be obtained from the following inequality:

$$T + n_c \times \tau \leq D \tag{4.17}$$

From Eq. (4.17), we conclude that the maximal number of checkpoints $n_{c_{max}}$ is calculated with the following expression:

$$n_{c_{max}} = \left\lfloor \frac{D-T}{\tau} \right\rfloor \tag{4.18}$$

As discussed earlier, for any $1 \leq n_c \leq n_{c_{max}}$, we can obtain the number of re-executions k that can take place without violating the deadline. We obtain the number of re-executions k from the following inequality:

$$T + n_c\tau + k \times (\frac{T}{n_c} + t) \leq D \tag{4.19}$$

By using the expression in Eq. (4.19), we compute k with the following expression:

$$k = \frac{D - (T + n_c \times \tau)}{\frac{T}{n_c} + \tau} \tag{4.20}$$

Note from the expression in Eq. (4.20) that for any $1 \leq n_c \leq n_{c_{max}}$ the number of re-executions k is always greater than or equal to zero. The denominator in Eq. (4.20) is always positive because T, τ, and n_c are all positive numbers. The numerator in Eq. (4.20) is positive for all $1 \leq n_c \leq n_{c_{max}}$, which directly follows from the inequality presented in Eq. (4.17). Hence, $k \geq 0$ for any $1 \leq n_c \leq n_{c_{max}}$.

As can be seen from Eq. (4.20), k is a function of n_c, and it is calculated by dividing the time interval $[{}^{n_c}t_0, D]$, where ${}^{n_c}t_0 = T + n_c \times \tau$, with the cost of a single re-execution, *i.e.* $\frac{T}{n_c} + \tau$. Increasing n_c from 1 to $n_{c_{max}}$, shrinks the time interval $[{}^{n_c}t_0, D]$ and reduces the cost of a single re-execution. Observe that the interval $[{}^{n_c}t_0, D]$ shrinks slowly, while the cost of a single re-execution reduces more rapidly as n_c increases. This indicates that k increases as n_c increases. However, increasing n_c beyond a certain value will result in a saturation of the cost of a single re-execution, while the interval $[{}^{n_c}t_0, D]$ will continuously shrink as n_c increases. Thus, increasing n_c beyond that certain value would decrease k. From this discussion, we conclude that there exists a maximal number of re-executions K_{max} that can take place without violating the deadline. Next, we derive the expression to compute K_{max}.

To obtain the maximum of the function given with Eq. (4.20), first we calculate the first derivative of Eq. (4.20) with respect to n_c, and then set it to be equal to zero.

$$\begin{aligned}
& \frac{\mathrm{d}}{\mathrm{d}n_c}\left(\frac{D-(T+n_c\times\tau)}{\frac{T}{n_c}+\tau}\right) = \\
= \quad & \frac{\mathrm{d}}{\mathrm{d}n_c}\left(n_c\times\frac{D-(T+n_c\times\tau)}{T+n_c\times\tau}\right) \\
= \quad & \frac{\mathrm{d}}{\mathrm{d}n_c}\left(\frac{n_c\times D}{T+n_c\times\tau}-n_c\right) \\
= \quad & \frac{\mathrm{d}}{\mathrm{d}n_c}\left(\frac{n_c\times D}{T+n_c\times\tau}\right)-\frac{\mathrm{d}}{\mathrm{d}n_c}\left(n_c\right) \\
\Rightarrow \quad & \frac{D\times(T+n_c\times\tau)-n_c\times D\times\tau}{(T+n_c\times\tau)^2}-1 \\
= \quad & \frac{D\times T}{(T+n_c\times\tau)^2}-1
\end{aligned} \tag{4.21}$$

The expression in Eq. (4.21) represents the first derivative of Eq. (4.20). The first derivative of Eq. (4.20) is equal to zero for a particular value of n_c which can be obtained by solving the following equation:

$$\frac{\mathrm{d}}{\mathrm{d}n_c}\left(\frac{D-(T+n_c\times\tau)}{\frac{T}{n_c}+\tau}\right)=0 \tag{4.22}$$

By using the expression in Eq. (4.21), solving the equation given in Eq. (4.22) results in the following:

$$\begin{aligned}
& \frac{D\times T}{(T+n_c\times\tau)^2}-1=0 \\
= \quad & \frac{D\times T-(T+n_c\times\tau)^2)}{(T+n_c\times\tau)^2}=0 \\
\Rightarrow \quad & \tau^2\times {n_c}^2+2\times T\times\tau\times n_c-D\times T+T^2=0 \\
\Rightarrow \quad & n_{c_{1,2}}=\frac{-2\times T\times\tau\pm\sqrt{4\times D\times T\times\tau^2}}{2\times\tau^2} \\
& n_{c_1}=-\frac{1}{\tau}\times(\sqrt{D\times T}+T) \qquad (4.23) \\
& n_{c_2}=\frac{1}{\tau}\times(\sqrt{D\times T}-T) \qquad (4.24)
\end{aligned}$$

As we can observe, solving equation Eq. (4.22) results in two solutions, *i.e.* n_{c_1} represented by Eq. (4.23) and n_{c_2} represented by Eq. (4.24). Note that n_{c_1}

has a negative value and as such it cannot be a valid solution (only positive integer values can be assigned to the number of checkpoints). Therefore, we claim that the function representing the number of re-executions k, given in (Eq. (4.20)), reaches its maximum at n_{c_2}. To ensure that the maximum of the function presented with Eq. (4.20) is obtained at n_{c_2}, we need to verify that the second derivative of Eq. (4.20) at n_{c_2} is negative. By taking the first derivative of Eq. (4.21), we calculate the second derivative of Eq. (4.20) as:

$$\begin{aligned}
& \frac{\mathrm{d}}{\mathrm{d}n_c}\left(\frac{D\times T}{(T+n_c\times\tau)^2}-1\right)= \\
= \quad & \begin{bmatrix} u = T + n_c \times \tau \\ \mathrm{d}u = \tau \times \mathrm{d}n_c \end{bmatrix} \\
= \quad & \frac{\tau}{\tau}\times\frac{\mathrm{d}}{\mathrm{d}n_c}\left(\frac{D\times T}{(T+n_c\times\tau)^2}-1\right) \\
= \quad & \tau\times\frac{\mathrm{d}}{\mathrm{d}u}\left(\frac{D\times T}{u^2}-1\right) \\
= \quad & \tau\times\frac{\mathrm{d}}{\mathrm{d}u}\left(\frac{D\times T}{u^2}\right)-\tau\times\frac{\mathrm{d}}{\mathrm{d}u}(1) \\
= \quad & -\frac{2\times\tau\times D\times T}{u^3} \\
= \quad & -\frac{2\times\tau\times D\times T}{(T+n_c\times\tau)^3}
\end{aligned} \tag{4.25}$$

Since the deadline D, the processing time T, and the checkpointing overhead τ are all positive numbers, the expression in Eq. (4.25) is negative for any positive n_c (including n_{c_2}). Hence, we have shown that the maximum of the function presented with Eq. (4.20) is obtained at n_{c_2}.

Next, we denote with $\hat{n}_c$ the number of checkpoints for which the function presented with Eq. (4.20) reaches its maximum, and as we have shown earlier in Eq. (4.24), it is calculated as:

$$\hat{n}_c = \frac{1}{\tau}\times(\sqrt{D\times T}-T) \tag{4.26}$$

Using Eq. (4.26), and by replacing n_c with $\hat{n}_c$ in Eq. (4.20), we calculate the maximum value $\hat{k}$ of the function representing the number of re-executions k with the following expression:

$$\hat{k} = \frac{D+T-2\times\sqrt{D\times T}}{\tau} \tag{4.27}$$

Note that both expressions presented in Eq. (4.26) and Eq. (4.27) are evaluated as real numbers, *i.e.* $\hat{k} \in \mathbb{R}$ and $\hat{n}_c \in \mathbb{R}$. However, in practice only

integer values are allowed for both n_c and k. Thus, the maximum number of re-executions, denoted with K_{max}, is evaluated as:

$$K_{max} = \left\lfloor \hat{k} \right\rfloor = \left\lfloor \frac{D + T - 2 \times \sqrt{D \times T}}{\tau} \right\rfloor \tag{4.28}$$

While the maximum value $\hat{k}$ of the function that represents the number of re-executions is obtained for a particular value for the number of checkpoints ($\hat{n}_c$), the maximum number of re-executions K_{max}, which is the closest rounded integer value lower than or equal to $\hat{k}$, can be obtained for a range of different n_c values. For each value of n_c that belongs to this range, the number of re-executions that can take place without violating the deadline is equal to K_{max}. We denote this range of values with ${}^{K_{max}}NC = [{}^{K_{max}}n_{c_L}, {}^{K_{max}}n_{c_U}]$, where ${}^{K_{max}}n_{c_L}$ and ${}^{K_{max}}n_{c_U}$ denote the lower and the upper bound, respectively. Important to note is that in the rest of the text we will use the following notations to address the lower and upper bounds of a range, *i.e.* $\lfloor {}^{K_{max}}NC \rfloor = {}^{K_{max}}n_{c_L}$ and $\lceil {}^{K_{max}}NC \rceil = {}^{K_{max}}n_{c_U}$. To calculate the bounds for ${}^{K_{max}}NC$, we need to solve the following equation:

$$\begin{aligned} D - \left(T + n_c \times \tau + K_{max} \times \left(\frac{T}{n_c} + \tau \right) \right) = 0 \\ \tau \times {n_c}^2 - (D - T - K_{max} \times \tau) \times n_c + K_{max} \times T = 0 \end{aligned} \tag{4.29}$$

By solving Eq. (4.29) we get:

$${}^{K_{max}}n_{c_l} = \frac{D - T - K_{max} \times \tau}{2 \times \tau} - \frac{\sqrt{(D - T - K_{max} \times \tau)^2 - 4 \times K_{max} \times T \times \tau}}{2 \times \tau} \tag{4.30}$$

$${}^{K_{max}}n_{c_u} = \frac{D - T - K_{max} \times \tau}{2 \times \tau} + \frac{\sqrt{(D - T - K_{max} \times \tau)^2 - 4 \times K_{max} \times T \times \tau}}{2 \times \tau} \tag{4.31}$$

Hence, for any $n_c \in [{}^{K_{max}}n_{c_l}, {}^{K_{max}}n_{c_u}]$, the number of re-executions that can take place without violating the deadline is equal to K_{max}. Observe that both ${}^{K_{max}}n_{c_l}$ and ${}^{K_{max}}n_{c_u}$, defined with Eq. (4.30) and Eq. (4.31) respectively,

are evaluated as real numbers, *i.e.* ${}^{K_{max}}n_{c_l} \in \mathbb{R}$ and ${}^{K_{max}}n_{c_u} \in \mathbb{R}$. However, the number of checkpoints is only allowed to be an integer number. Therefore, the bounds of the range ${}^{K_{max}}NC$ are calculated as:

$${}^{K_{max}}n_{c_L} = \left\lceil {}^{K_{max}}n_{c_l} \right\rceil = \left\lceil \frac{D - T - K_{max} \times \tau}{2 \times \tau} - \frac{\sqrt{(D - T - K_{max} \times \tau)^2 - 4 \times K_{max} \times T \times \tau}}{2 \times \tau} \right\rceil \tag{4.32}$$

$${}^{K_{max}}n_{c_U} = \left\lfloor {}^{K_{max}}n_{c_u} \right\rfloor = \left\lfloor \frac{D - T - K_{max} \times \tau}{2 \times \tau} + \frac{\sqrt{(D - T - K_{max} \times \tau)^2 - 4 \times K_{max} \times T \times \tau}}{2 \times \tau} \right\rfloor \tag{4.33}$$

Observe that for any $n_c < \left\lfloor {}^{K_{max}}NC \right\rfloor$ the number of re-executions k is strictly lower than K_{max}, and further k increases as n_c increases from 1 to $\left\lfloor {}^{K_{max}}NC \right\rfloor$. On the other hand, for any $n_c > \left\lceil {}^{K_{max}}NC \right\rceil$ the number of re-executions k is again strictly lower than K_{max}. However, k decreases as n_c increases from $\left\lceil {}^{K_{max}}NC \right\rceil$ to $n_{c_{max}}$. This implies that the same number of re-executions can be achieved for a discontinuous range of values of n_c, *i.e.* a range of values to the left of ${}^{K_{max}}NC$ and a range of values to the right of ${}^{K_{max}}NC$. We denote these discontinuous ranges with ${}^{k}NC$, where the index k belongs to the range $[0, K_{max})$ (observe that the upper bound is not included in the range). Important to note about a discontinuous range ${}^{k}NC$ is that for any $n_c \in {}^{k}NC$ the number of re-executions that can take place without violating the deadline is exactly equal to the index k in the notation ${}^{k}NC$. Next, we elaborate on how to evaluate ${}^{k}NC$.

The analysis is performed by assuming that given is the number of re-executions $k^\dagger$ for which we want to identify ${}^{k^\dagger}NC$ and $k^\dagger \in [0, K_{max})$. For the given $k^\dagger$, we are only interested in those values of n_c that would not violate the given deadline D, and we can express that statement with the following inequality:

$$T + n_c \times \tau + k^\dagger \times (\frac{T}{n_c} + \tau) \leq D \tag{4.34}$$

The expression in Eq. (4.34) can be re-written as:

$$\tau \times n_c^2 + (T - D + k^\dagger \times \tau) \times n_c + k^\dagger \times T \leq 0 \tag{4.35}$$

The left-hand side of the expression given in Eq. (4.35) represents a quadratic function with respect to n_c that reaches a minimum point (the coefficient in front of n_c^2 is τ which is positive). Such a quadratic function can have negative values if and only if the roots of the quadratic function are two different real numbers n_{c_1} and n_{c_2}, in which case the quadratic function will be evaluated as negative for all $n_c \in (n_{c_1}, n_{c_2})$. Thus, to find the range of values of n_c for which the inequality given in Eq. (4.35) holds, it is necessary to solve the following equation:

$$\tau \times n_c^2 + (T - D + k^\dagger \times \tau) \times n_c + k^\dagger \times T = 0 \tag{4.36}$$

The solutions of Eq. (4.36) are:

$$^{k^\dagger}n_{c_L} = \left\lceil \frac{D - T - k^\dagger \times \tau}{2 \times \tau} - \frac{\sqrt{(D - T - k^\dagger \times \tau)^2 - 4 \times k^\dagger \times T \times \tau}}{2 \times \tau} \right\rceil \tag{4.37}$$

$$^{k^\dagger}n_{c_U} = \left\lfloor \frac{D - T - k^\dagger \times \tau}{2 \times \tau} + \frac{\sqrt{(D - T - k^\dagger \times \tau)^2 - 4 \times k^\dagger \times T \times \tau}}{2 \times \tau} \right\rfloor \tag{4.38}$$

It is important to note that $^{k^\dagger}n_{c_L}$ and $^{k^\dagger}n_{c_U}$, in Eq. (4.37) and Eq. (4.38) respectively, depend on $k^\dagger$. The inequality in Eq. (4.35) holds for any n_c that belongs to $[^{k^\dagger}n_{c_L}, ^{k^\dagger}n_{c_U}]$, and it states that for each n_c in the range $[^{k^\dagger}n_{c_L}, ^{k^\dagger}n_{c_U}]$ *at least* $k^\dagger$ re-executions can take place without violating the deadline D. However, it does not exclude the fact that for some n_c values, which belong to the same range, it is possible to have more than $k^\dagger$ re-executions that can take place without violating the deadline. Since our goal is to find the range of values of n_c for which *exactly* $k^\dagger$ re-executions can take place without violating the deadline, we need to exclude all those n_c values for which *at least* $k^\dagger + 1$ re-executions can take place without violating the deadline. The range of n_c values for which *at least* $k^\dagger + 1$ re-executions can take place without violating the deadline is $[^{k^\dagger+1}n_{c_L}, ^{k^\dagger+1}n_{c_U}]$, where $^{k^\dagger+1}n_{c_L}$ and $^{k^\dagger+1}n_{c_U}$ can be calculated by evaluating the equations Eq. (4.37) and Eq. (4.38) respectively

at $k^\dagger + 1$. Finally, the discontinuous range of n_c values for which *exactly* $k^\dagger$ re-executions can take place without violating the deadline is:

$$^{k^\dagger}NC = [^{k^\dagger}n_{c_L}, ^{k^\dagger+1}n_{c_L}) \cup (^{k^\dagger+1}n_{c_U}, ^{k^\dagger}n_{c_U}],\ 0 \leq k^\dagger < K_{max} \tag{4.39}$$

Observe that Eq. (4.37), Eq. (4.38) and Eq. (4.39) were derived while assuming a particular value $k^\dagger$ for the number of re-executions k, and therefore the notation $k^\dagger$ is present in all equations. However, the equations are valid for any value of k that belongs to the range $[0, K_{max}]$, and therefore replacing the notation $k^\dagger$ with k in Eq. (4.37), Eq. (4.38), and Eq. (4.39) is justified.

From the presented discussion, we observe that the range $[1, n_{c_{max}}]$ of n_c values can be divided into $K_{max} + 1$ different ranges ^kNC defined as:

$$^kNC = \begin{cases} [^{K_{max}}n_{c_L}, ^{K_{max}}n_{c_U}] & k = K_{max} \\ [^kn_{c_L}, ^{k+1}n_{c_L}) \cup (^{k+1}n_{c_U}, ^kn_{c_U}] & 0 \leq k < K_{max} \end{cases}$$

A given n_c can belong to only one ^kNC, and for all the n_c values that belong to the same ^kNC, the number of re-executions that can take place without violating the deadline is the same, *i.e.* it is equal to k. When evaluating the LoC with respect to a given deadline D, for any $n_c \in {}^kNC$, the expression that computes $\Lambda_{n_c}(D)$ includes $k + 1$ terms from the probability distribution function. This is shown with the following expression:

$$\Lambda_{n_c}(D) = \sum_{i=0}^{k} p_{n_c}(i),\ n_c \in {}^kNC \tag{4.40}$$

For any given n_c that belongs to ^kNC, $\Lambda_{n_c}(D)$ is exactly the same as $\lambda_{n_c}(k)$. We have already shown in **Theorem 3** that for a fixed number of re-executions k, $\lambda_{n_c}(k)$ increases with n_c. Since for all n_c that belong to ^kNC the number of re-executions is equal to k, according to **Theorem 3**, the highest LoC among all $n_c \in {}^kNC$ will be obtained for the upper bound the range ^kNC, *i.e.* $n_c = \left\lceil {}^kNC \right\rceil$.

As the goal is to obtain the maximal LoC, for a given range ^kNC, there is no need to explore $\Lambda_{n_c}(D)$ for all $n_c \in {}^kNC$. Instead, it is sufficient to explore $\Lambda_{n_c}(D)$ only for the n_c value that represents the upper bound of the range ^kNC, *i.e.* $n_c = \left\lceil {}^kNC \right\rceil$.

Since there are a total of $K_{max} + 1$ different ranges ^kNC ($0 \leq k \leq K_{max}$), and due to the fact that it is sufficient to explore $\Lambda_{n_c}(D)$ only for one n_c value per range, to obtain the optimal number of checkpoints which results in the maximal LoC it is sufficient to compute $\Lambda_{n_c}(D)$ for a total of $K_{max} + 1$ different n_c values. These $K_{max} + 1$ values of n_c represent the upper bounds

of ${}^{k}NC,\ 0 \leq k \leq K_{max}$. Observe that the value $\lceil {}^{0}NC \rceil$, *i.e.* $\left\lceil {}^{k}NC \right\rceil$ for $k = 0$, is also included among the $K_{max} + 1$ different n_c values. However, we show next that there is no need to explore this value as long as K_{max} is greater than one.

As stated in **Theorem 1**, if no re-executions can take place without violating the deadline, *i.e.* n_c belongs to ${}^{0}NC$, $\Lambda_{n_c}(D)$ is equal to ${P_T}^2$ for any $n_c \in {}^{0}NC$. Therefore, if $K_{max} = 0$, any n_c value in the range $[1, n_{c_{max}}]$ would provide the same $\Lambda_{n_c}(D)$, *i.e.* ${P_T}^2$. However, if $K_{max} > 0$, a higher LoC than ${P_T}^2$ can be achieved for any $k^{\dagger}$ that is greater than or equal to one. This can be shown by simply re-writing Eq. (4.9) with the expression given in Eq. (4.41), while assuming that $k^{\dagger}$ is greater than or equal to one.

$$\begin{aligned}\Lambda_{n_c}(D) &= \sum_{k=0}^{k^{\dagger}} p_{n_c}(k) \\ &= p_{n_c}(0) + \sum_{k=1}^{k^{\dagger}} p_{n_c}(k) \\ &= {P_T}^2 + \sum_{k=1}^{k^{\dagger}} p_{n_c}(k)\end{aligned} \tag{4.41}$$

As shown in Eq. (4.41), $\Lambda_{n_c}(D)$ is strictly greater than ${P_T}^2$ whenever $k^{\dagger} \geq 1$. This discussion justifies that there is no need to compute the LoC for $\lceil {}^{0}NC \rceil$. Instead, it is sufficient to compute $\Lambda_{n_c}(D)$ for K_{max} different n_c values, and these values are the upper bounds of ${}^{k}NC,\ 1 \leq k \leq K_{max}$.

Finally, the conclusion is that the maximal LoC with respect to a given deadline is obtained as:

$$\Lambda_{n_c^*}(D) = MAX\left\{\Lambda_{n_c}(D),\ n_c = \left\lceil {}^{k}NC \right\rceil\ k = 1, 2...K_{max}\right\} \tag{4.42}$$

The maximal LoC is obtained at a specific $n_c^k,\ k \in [1, K_{max}]$, computed as:

$$\begin{aligned}n_c^k &= \left\lceil {}^{k}NC \right\rceil \\ &= \left\lfloor \frac{D - T - k \times \tau}{2 \times \tau} \right. \\ &\quad \left. + \frac{\sqrt{(D - T - k \times \tau)^2 - 4 \times k \times T \times \tau}}{2 \times \tau} \right\rfloor\end{aligned} \tag{4.43}$$

Important to note is that if K_{max} is equal to zero, then the maximal LoC is equal to ${P_T}^2$, and it can be achieved for any $n_c \in [1, n_{c_{max}}]$.

Scenario A	Scenario B
T = 1000 t.u.	T = 1000 t.u.
τ = 20 t.u.	τ = 20 t.u.
P_T = 0.99999	P_T = 0.9

Table 4.1.: Input scenarios

4.1.5. EXPERIMENTAL RESULTS

The purpose of the experiments presented in this section is twofold. First, we show that the LoC varies with the number of checkpoints, and we validate our approach that finds the optimal number of checkpoints that results in the maximal LoC. Second, we show that optimizing RRC towards soft RTSs results in poor probabilistic guarantees of meeting the deadlines, *i.e.* low LoC with respect to the given deadline or low LoC with respect to the minimal AET.

For that purpose, we present results for the following experiments:

- P1: evaluation of the LoC with respect to a given deadline D;
- P2: evaluation of probabilistic guarantees when RRC is optimized for soft RTSs. For this experiment, we consider the following two cases:
  - P2A: evaluation of the LoC with respect to the minimal AET, and
  - P2B: evaluation of the LoC with respect to a given deadline D, when n_c is optimized towards AET.

For each experiment, we use two input scenarios, Scenario A and Scenario B, which are summarized in Table 4.1. For each scenario, the following inputs are given: the processing time of a job T, the checkpointing overhead τ, and the probability P_T that no errors occur in a processor within an interval equal to T. Additionally, for P1 and P2B, where we evaluate the LoC with respect to a deadline, we assume given is a deadline $D = 1500$ time units (t.u.).

For P1, *i.e.* evaluation of LoC with respect to a given deadline D, the results are presented in Table 4.2 and Table 4.3. In Table 4.2 and Table 4.3, we show the computed LoC with respect to the given deadline D, *i.e.* $\Lambda_{n_c}(D)$, at various number of checkpoints n_c. The results are obtained as follows. For each n_c, we first calculate the number of re-executions K that can take place without violating the deadline, and then we sum all terms from the probability distribution function $p_{n_c}(k)$ (see Eq. (4.6)) for $k \in [0, K]$. The objective of this experiment is to show that (1) the LoC with respect to the given deadline

varies with the number of checkpoints n_c, and (2) the optimization method discussed in Section 4.1.4 finds the optimal number of checkpoints that results in the maximal LoC.

As can be seen from Table 4.2 and Table 4.3, the LoC with respect to the given deadline D depends on the number of checkpoints n_c. When the number of checkpoints is low, the LoC is also low. The LoC increases as the number of checkpoints increases. The increase in LoC is either done by subtle increments or step-wise increments (observe the LoC for $n_c \in [1, 17]$ in Table 4.2 and Table 4.3). The subtle increments can be observed when n_c changes from 3 to 5 and when n_c changes from 6 to 17, while the step-wise increments are observed when n_c changes from 2 to 3 and when n_c changes from 5 to 6. Increasing the number of checkpoints above a certain point causes the step-wise increments in the LoC to be replaced by sharp drops (observe the LoC when n_c changes from 17 to 18 and when n_c changes from 21 to 22 in both Table 4.2 and Table 4.3). However, increasing the number of checkpoints beyond a value for which a sharp drop in the LoC has occurred, would increase the LoC (observe the LoC when n_c changes from 18 to 21 in both Table 4.2 and Table 4.3). However, the recent increase is only done by subtle increments. This trend of subtle increments followed by sharp drops proceeds as the number of checkpoints increases. Finally, increasing the number of checkpoints above a certain point leads to an LoC equal to zero. The reason for this behavior is the following. When the number of checkpoints is low, the execution segments are longer, which means that only a limited number of re-executions can take place without violating the deadline. This implies that only a small number of terms from the probability distribution function (Eq. (4.6)) will be summed, and therefore the LoC (Eq. (4.8)) is low. Increasing the number of checkpoints, decreases the length of the execution segments, and thus allows more re-executions to take place without violating the deadline on one hand, but increases the total checkpointing overhead on the other hand. Each time an extra term is added (one more re-execution can take place without violating the deadline), results in a step-wise increase in the LoC. However, after a certain point the total checkpointing overhead becomes very high, and it reduces the number of re-executions that can take place without violating the deadline. Whenever such scenario occurs, it results in a sharp drop in the LoC. The subtle increments in the LoC are a consequence of **Theorem 3**. When the number of checkpoints is excessively high, the overhead due to taking the checkpoints may violate the deadline, *i.e.* LoC equal to zero. As ${}^{n_c}t_0$, the case when zero erroneous execution segments are executed, depends on the number of checkpoints n_c (see Eq. (4.1)), using many checkpoints may result in that ${}^{n_c}t_0$ violates the deadline D, *i.e.* ${}^{n_c}t_0 > D$. For example, for the given input scenarios when $n_c = 26$, ${}^{26}t_0 = 1000 + 26 \times 20 = 1520$ t.u., and thus $\Lambda_{26}(D) = 0$ (see Table 4.2 and Table 4.3). With the results obtained from

$D = 1500$			
n_c	$\Lambda_{n_c}(D)$	n_c	$\Lambda_{n_c}(D)$
1	0.999980000100000000	14	0.999999999999998367
2	0.999980000100000000	15	0.999999999999998388
3	0.999999999733334814	16	0.999999999999998406
4	0.999999999750001250	**17**	**0.999999999999998422**
5	0.999999999760001120	18	0.999999999788889670
6	0.999999999999997925	19	0.999999999789474459
7	0.999999999999998040	20	0.999999999790000770
8	0.999999999999998125	21	0.999999999790476955
9	0.999999999999998189	22	0.999980000100000000
10	0.999999999999998240	23	0.999980000100000000
11	0.999999999999998280	24	0.999980000100000000
12	0.999999999999998314	25	0.999980000100000000
13	0.999999999999998343	26	0

Table 4.2.: $\Lambda_{n_c}(D)$ for different n_c values, for Scenario A

$D = 1500$			
n_c	$\Lambda_{n_c}(D)$	n_c	$\Lambda_{n_c}(D)$
1	0.810000000000000000	14	0.998386333221060871
2	0.810000000000000000	15	0.998405709197021325
3	0.974827503159636872	16	0.998422589149847735
4	0.976266114316335439	**17**	**0.998437425722750770**
5	0.977137362167560214	18	0.979688847172390437
6	0.997980204415657095	19	0.979741032210778210
7	0.998085015474654920	20	0.979788017059326005
8	0.998162202793752259	21	0.979830542116846522
9	0.998221387037794418	22	0.810000000000000000
10	0.998268194669895683	23	0.810000000000000000
11	0.998306132813719019	24	0.810000000000000000
12	0.998337499909652013	25	0.810000000000000000
13	0.998363864473716882	26	0

Table 4.3.: $\Lambda_{n_c}(D)$ for different n_c values, for Scenario B

solving P1, we want to point out that it is useful to have a framework to calculate the LoC because it makes it possible to optimize the RRC scheme such that the optimal number of checkpoints that results in the maximal LoC can be obtained. From the results presented in Table 4.2 and Table 4.3, we note that the number of checkpoints that provides the maximal LoC is $n_c = 17$ for both Scenario A and Scenario B. However, $\Lambda_{n_c}(D)$ for Scenario A is much higher than $\Lambda_{n_c}(D)$ for Scenario B due to the different values used for P_T.

In Section 4.1.4, we proposed a method to obtain the optimal number of checkpoints that maximizes the LoC of meeting a given deadline. Next, we verify that the results obtained by using this method adhere to the results that we have presented in Table 4.2 and Table 4.3. The method suggests that it is sufficient to examine the LoC for only K_{max} different n_c values which can be obtained by using Eq. (4.43) for $k \in [1, K_{max}]$. First, we evaluate the maximal number of re-executions K_{max} that can take place without violating the deadline D. The expression for calculating K_{max} is presented in Eq. (4.28). As we can observe from Eq. (4.28), K_{max} depends on the following parameters: the deadline D, the processing time T, and the checkpointing overhead τ. Observe that all these parameters are the same for both Scenario A and Scenario B, *i.e.* $D = 1500$ t.u., $T = 1000$ t.u., and $\tau = 20$ t.u. Using these inputs in Eq. (4.28), we evaluate K_{max} as:

$$K_{max} = \left\lfloor \frac{1500 + 1000 - 2 \times \sqrt{1500 \times 1000}}{2 \times 20} \right\rfloor = 2$$

Since $K_{max} = 2$, we need to examine the LoC for only two values of n_c. As we mentioned earlier in Section 4.1.4, those values can be obtained by using Eq. (4.43) for $k \in [1, K_{max}]$. By evaluating Eq. (4.43) for $k \in [1, 2]$, using the given inputs, we get:

$$n_c^1 = \left\lfloor \frac{1500 - 1000 - 1 \times 20}{2 \times 20} + \frac{\sqrt{(1500 - 1000 - 1 \times 20)^2 - 4 \times 2 \times 1000 \times 20}}{2 \times 20} \right\rfloor = 21$$

$$n_c^2 = \left\lfloor \frac{1500 - 1000 - 2 \times 20}{2 \times 20} + \frac{\sqrt{(1500 - 1000 - 2 \times 20)^2 - 4 \times 2 \times 1000 \times 20}}{2 \times 20} \right\rfloor = 17$$

Since the deadline D, the processing time T, and the checkpointing overhead τ are the same for both Scenario A and Scenario B, it leads to the fact that $n_c^k,\ k\ \in [1, K_{max}]$, obtained using Eq. (4.43), will be the same for Scenario A and Scenario B.

By comparing the LoC for n_c^1 and n_c^2, for both Scenario A and Scenario B, we conclude that a higher LoC is achieved for n_c^2. Thus, the optimal number of checkpoints that results in the maximal LoC for both Scenario A and Scenario B is $n_c^* = n_c^2 = 17$, which can also be seen in Table 4.2 and Table 4.3.

Observe that the presented results may mislead the reader into drawing the conclusion that the optimal number of checkpoints n_c^* which results in the maximal LoC is always equal to $n_c^{K_{max}}$. This in turn would mean that there exists a closed-form expression to compute the optimal number of checkpoints that results in the maximal LoC. However, we want to point out and clarify that even though in many cases it can happen that $n_c^* = n_c^{K_{max}}$, in general, n_c^* can be evaluated only after comparing the LoC for all $n_c^k\ k \in [1, K_{max}]$. Next, we present the results for the second experiment P2.

For P2, *i.e.* evaluation of probabilistic guarantees when RRC is optimized for soft RTSs, we consider the RRC optimization approach discussed in Chapter 3, where we obtain the optimal number of checkpoints n_c^* that leads to the minimal AET. Applying the equation for obtaining the optimal number of checkpoints (see Eq. (3.20) in Section 3.3), we get the following results:

- For Scenario A, we compute the optimal number of checkpoints $n_c^* = 1$ which provides the minimal AET=1020 t.u., and
- For Scenario B, we compute the optimal number of checkpoints $n_c^* = 3$ which provides the minimal AET=1138 t.u.

The purpose of this experiment is to show that optimizing RRC with respect to AET results in poor probabilistic guarantees that the job completes before the minimal AET, or before a given deadline.

For P2A, *i.e.* evaluation of $\Lambda_{n_c}(\text{AET})$, we evaluate the LoC with respect to the minimal AET, while assuming that the number of checkpoints used is equal to the optimal number of checkpoints n_c^* that results in the minimal AET. By computing the LoC for the calculated minimal AET, we observe that $\Lambda_{n_c}(1020) = 0.99998$ for Scenario A, and $\Lambda_{n_c}(1138) = 0.81$ for Scenario B, which may be acceptable for a soft RTS, but not for a hard RTS where a very high LoC is required.

For P2B, *i.e.* evaluation of $\Lambda_{n_c}(D)$ when n_c is optimized towards AET, we present the LoC with respect to the given deadline, while considering that the number of checkpoints is chosen such that minimal AET is achieved. As shown earlier, we have computed the optimal number of checkpoints, *i.e.*

$n_c^* = 1$ for Scenario A and $n_c^* = 3$ for Scenario B, that minimizes the AET. Relying on this optimization implies the following results:

- for Scenario A, $\Lambda_{n_c}(D) = 0.99998$ (see Table 4.2 for $n_c = 1$)
- for Scenario B, $\Lambda_{n_c}(D) \leq 0.975$ (see Table 4.3 for $n_c = 3$).

However, we observed earlier (see Table 4.2 and Table 4.3) that the highest LoC that can be achieved is:

- for Scenario A, $\Lambda_{n_c}(D) = 0.999999999999998422$ for $n_c = 17$
- for Scenario B, $\Lambda_{n_c}(D) \geq 0.99843$ for $n_c = 17$.

From the results presented for P2 (P2A and P2B), we conclude that relying on RRC optimization for soft RTSs results in poor probabilistic guarantees. In particular, we showed that the LoC with respect to the minimal AET is quite low, which in turn shows that the probability that a job takes longer time than the expected minimal AET is not negligible (observe the results for P2A). Further, we showed that if the number of checkpoints is selected such that the minimal AET is achieved, it results in a much lower LoC in comparison to the maximal LoC that can be achieved for a different value of the number of checkpoints (observe the results for P2B).

4.2. MULTIPLE JOBS

We showed in the previous section how to find the optimal number of checkpoints that maximizes the LoC with respect to a given deadline for a single job. In this section, we extend the problem to address multiple jobs. The extended problem consists of finding the optimal checkpoint assignment, *i.e.* the number of checkpoints to be used by each job such that the LoC with respect to a given global deadline is maximized.

While the solution to this problem seems to be a superset of the solution for the single job problem, we show in this section that the problem with multiple jobs is much more complex. The main issues we discuss in this section are as follows:

- We show that performing a local optimization for each job and combining these local optima together does not result in the maximal LoC with respect to the global deadline.
- We show that handling the set of jobs as a one single large job and obtaining the optimal number of checkpoints for the single large job does not result in the maximal LoC with respect to the global deadline.

- We provide an expression to evaluate the LoC with respect to a given global deadline for a given checkpoint assignment, where the checkpoint assignment defines the number of checkpoints to be used by each job in the given set of jobs.
- We show that a holistic solution (exhaustive search over all possible checkpoint assignments) is required to obtain the optimal checkpoint assignment and the maximal LoC.
- We propose a method (heuristic) to speed up the computations and obtain the results in significantly shorter time compared to the exhaustive search method.
- We present experimental results to demonstrate that our method is capable of finding the optimal checkpoint assignment and the maximal LoC while observing tremendous reduction in computation time compared to the exhaustive search method.

The rest of this section is organized as follows. We state the problem formulation in Section 4.2.1. In Section 4.2.2 we review two approaches that aim to solve the problem by directly applying the solution for the single job problem. In Section 4.2.3, we first present a mathematical expression to evaluate the LoC for a given checkpoint assignment, and then we outline a method, *i.e.* Exhaustive Search, that finds the maximal LoC. In Section 4.2.4 we propose a method that aims to solve the problem optimally at a significantly lower computational cost. Finally, experimental results are presented in Section 4.2.5.

4.2.1. PROBLEM FORMULATION

In this section, we analyze the LoC with respect to a given global deadline for a set of jobs. The problem is described as follows. Given the following inputs:

- a set of m jobs, where each job has a processing time T,
- a checkpointing overhead τ,
- a global deadline D, and
- a probability P_T that no errors occur in a processing node within an interval of length T.

find the optimal checkpoint assignment $\bar{n}_c^* = [n_{c_1}, n_{c_2}, ...n_{c_m}]$, *where* $\bar{n}_c^*$ *represents a vector where each element* n_{c_i} *defines the number of checkpoints for each job, that results in the maximal LoC with respect to the global deadline* D.

Important to note is that in the rest of the text, we use the notation $\Lambda_{\bar{n}_c}(D)$ whenever we address the LoC for multiple jobs using the checkpoint assignment $\bar{n}_c$, while we use the notation $\Lambda_{n_c}(D)$ whenever we address the LoC for a single job using n_c checkpoints.

4.2.2. MOTIVATION

In this section, we investigate two approaches that apply the solution for a single job to solve the problem for a set of jobs. The approaches are:

1. **Local Optimization:** perform local optimization for a single job and apply the results to all jobs, *i.e.* find the optimal number of checkpoints for one job such that the LoC with respect to a local deadline is maximal and apply this optimal number of checkpoints to all jobs.

2. **Single Large Job:** assume that the set of jobs is equal to one single large job and find the optimal number of checkpoints for this job, such that the LoC with respect to the global deadline is maximal.

The models for these two approaches along with the model for the stated problem formulation are illustrated in Figure 4.4. The rest of this section details each of the approaches.

LOCAL OPTIMIZATION The basic idea is to perform optimization for a single job and apply the results to all jobs. The model for the Local Optimization approach is illustrated in Figure 4.4(b). As shown in Figure 4.4(b), this model requires the introduction of local deadlines. Instead of observing a set of jobs running in a sequence and a global deadline D, the model suggests a set of jobs, where each job has its own local deadline D'. Since all jobs have the same processing time T, the local deadline can be calculated as $D' = \frac{D}{m}$ where D denotes the global deadline, and m denotes the number of jobs in the set. This approach computes the optimal number of checkpoints that results in the maximal LoC with respect to the local deadline D'. A consequence of this approach is that the same number of checkpoints is assigned to all jobs in the set. It means that the resulting checkpoint assignment is represented as a vector where all elements have the same value, *i.e.* $\bar{n}_c = [n_{c_1}, n_{c_2}, ...n_{c_m}]$ where $n_{c_i} = n_{c_j} \; \forall \, i, j \in [1, m]$.

The first step is to obtain the optimal number of checkpoints n_c^* for a single job that results in the maximal LoC with respect to the local deadline D'. The optimal number of checkpoints n_c^* is obtained as shown in Section 4.1.4. The next step is to calculate the LoC with respect to the global deadline D, *i.e.* $\Lambda_{\bar{n}_c^*}(D)$, while considering that a checkpoint assignment $\bar{n}_c^* = [n_c^*, n_c^*, ...n_c^*]$ is

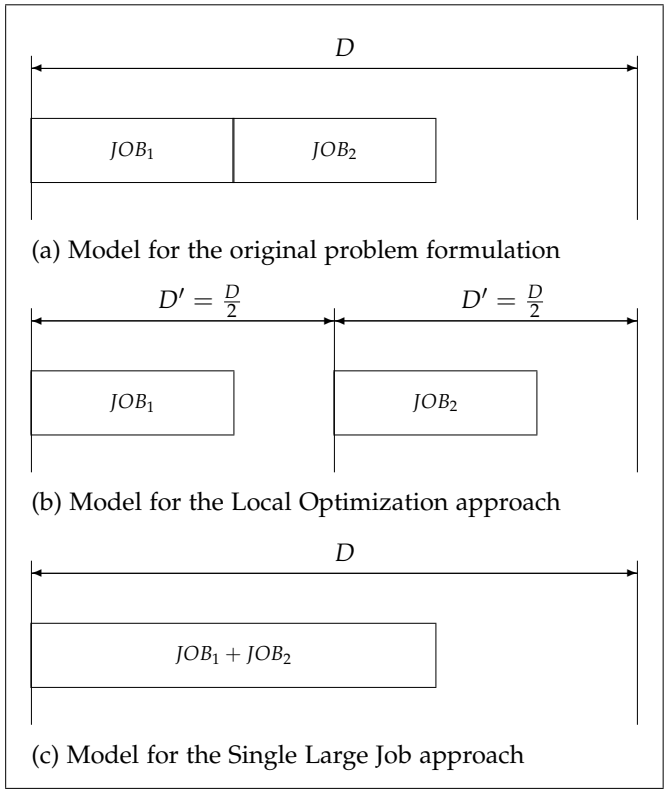


Figure 4.4.: Illustration of models used for (a) stated problem formulation, (b) Local Optimization, and (c) Single Large Job approach

used. One solution to calculate $\mathbb{\Lambda}_{\bar{n}_c^*}(D)$ is given in Eq. (4.44).

$$\mathbb{\Lambda}_{\bar{n}_c^*}(D) = \prod_{i=1}^{m} \Lambda_{n_c^*}(D') = \Lambda_{n_c^*}(D')^m \tag{4.44}$$

However, the expression in Eq. (4.44) does not calculate the correct $\mathbb{\Lambda}_{\bar{n}_c^*}(D)$ for the given checkpoint assignment $\bar{n}_c^*$. To show that Eq. (4.44) does not calculate the correct $\mathbb{\Lambda}_{\bar{n}_c^*}(D)$ consider the following scenario. Let us assume a scenario of two jobs with a processing time T and a global deadline D. We assume that local deadlines $D' = \frac{D}{2}$ are introduced and n_c^* is the optimal number of checkpoints that results in the maximal LoC for a single job with respect to the local deadline, *i.e.* $\Lambda_{n_c^*}(D')$. In such scenario, for each job, there is a limited number of re-executions k^* that can take place without violating

the local deadline D', and $\Lambda_{n_c^*}(D')$ is calculated as:

$$\Lambda_{n_c^*}(D') = \sum_{k=0}^{k^*} p_{n_c^*}(k) \tag{4.45}$$

According to Eq. (4.44), for the scenario of two jobs where the checkpoint assignment $\bar{n}_c^* = [n_c^*, n_c^*]$ is used, $\mathbb{\Lambda}_{\bar{n}_c^*}(D)$ is evaluated as:

$$\mathbb{\Lambda}_{\bar{n}_c^*}(D) = \Lambda_{n_c^*}(D') \times \Lambda_{n_c^*}(D') \tag{4.46}$$

Replacing the expression for $\Lambda_{n_c^*}(D')$, given in Eq. (4.45), in Eq. (4.46), we get:

$$\mathbb{\Lambda}_{\bar{n}_c^*}(D) = \sum_{k=0}^{k^*} p_{n_c^*}(k) \times \sum_{k=0}^{k^*} p_{n_c^*}(k) \tag{4.47}$$

From Eq. (4.47) one observes that $\mathbb{\Lambda}_{\bar{n}_c^*}(D)$ is calculated while considering that each job can have at most k^* re-executions (erroneous execution segments). According to its definition, the LoC is the probability that a job (or a set of jobs) completes *before* a given deadline. Such probability is calculated as a sum of terms, where each term represents the probability that a job (or a set of jobs) completes *exactly at* a given discrete time moment. Thus, obtaining the LoC requires summing the probabilities for *all* discrete time moments prior to the given deadline. Not including *all* terms in the sum, results in incorrect LoC which is the drawback of Eq. (4.44). Since the expression in Eq. (4.47), derived from Eq. (4.44), considers that each job can have at most k^* re-executions, *all* terms are not included in the sum that calculates the LoC. For example, if the first job has completed without any re-executions, then the second job may tolerate more than k^* re-executions ($2 \times k^*$ re-executions in this example) before the global deadline D. However, the expression in Eq. (4.47) considers k^* re-executions for the second job even if the first one has completed without any re-executions. With this discussion we justify that the expression in Eq. (4.44) does not calculate the correct LoC with respect to the given global deadline when the checkpoint assignment $\bar{n}_c^* = [n_c^*, n_c^* ... n_c^*]$ is used. Before we derive an expression to calculate $\mathbb{\Lambda}_{\bar{n}_c^*}(D)$, we want to point out that the expression in Eq. (4.44) would be correct as long as the execution of the jobs is scheduled such that a job starts with its execution immediately after the local deadline of its predecessor (as illustrated in Figure 4.4(b)). In such schedule, each job must complete before its own (local) deadline, and even if a job completes before its deadline, the next job will not start its execution immediately after the completion of its predecessor. However, according to the stated problem formulation (see Figure 4.4(a)) a job starts with its execution immediately after the completion of its predecessor.

Since Eq. (4.44) does not calculate the correct $\mathbb{A}_{\bar{n}_c^*}(D)$, in the rest of this section, we derive an expression on how to calculate $\mathbb{A}_{\bar{n}_c^*}(D)$ when a checkpoint assignment $\bar{n}_c^* = [n_c^*, n_c^* ... n_c^*]$ is used. First, we derive the expression for a scenario of two jobs, and then we provide the expression for a general scenario that consists of m jobs.

Let us assume a scenario where given is a global deadline D and two jobs, each with a processing time T. First, for a single job we compute the optimal number of checkpoints n_c^* by using the notion of local deadlines. Once n_c^* is obtained, n_c^* checkpoints are assigned to both jobs. That means we consider a checkpoint assignment $\bar{n}_c^* = [n_c^*, n_c^*]$. For such checkpoint assignment, the jobs can complete only at discrete time moments, which are computed according to the following expression:

$$^{\bar{n}_c^*}t_k = 2 \times (T + n_c^* \times \tau) + k \times (\frac{T}{n_c^*} + \tau) \tag{4.48}$$

The first term in Eq. (4.48) calculates the minimum required time for both jobs to complete, *i.e.* each job needs to spend at least T time units on execution along with $n_c^* \times \tau$ time units that come from the checkpointing overhead of taking n_c^* checkpoints. The second term in Eq. (4.48) calculates the extra time that is spent on execution of erroneous execution segments. The index k in the expression for $^{\bar{n}_c^*}t_k$, presented in Eq. (4.48), denotes the total number of erroneous execution segments. For the given scenario there might be a number of erroneous execution segments during the execution of the first job, which we denote with k_1, and a number of erroneous execution segments during the execution of the second job, which we denote with k_2. Thus, the total number of erroneous execution segments is evaluated as $k = k_1 + k_2$.

To calculate the LoC with respect to the global deadline, we sum the probabilities that both jobs complete at any time moment $^{\bar{n}_c^*}t_k \leq D$. For this reason, we need a probability distribution function $p'_{\bar{n}_c^*}(k)$ which calculates the probability that both jobs complete at a time moment $^{\bar{n}_c^*}t_k$, and this function is defined as:

$$p'_{\bar{n}_c^*}(k) = \sum_{i=0}^{k} p_{n_c^*}(i) \cdot p_{n_c^*}(k-i) \tag{4.49}$$

The expression in Eq. (4.49) is derived as a sum of intermediate terms, where each term represents the combined probability that i erroneous execution segments are executed during the execution of the first job, *i.e.* probability of having i erroneous execution segments $p_{n_c^*}(i)$, and $k-i$ erroneous execution segments are executed during the execution of the second job, which is denoted as $p_{n_c^*}(k-i)$. Using Eq. (4.6), we re-write the expression for calculating the probability distribution function $p'_{\bar{n}_c^*}(k)$ as:

$$
\begin{aligned}
p'_{\bar{n}_c^*}(k) &= \sum_{i=0}^{k} p_{n_c^*}(i) \times p_{n_c^*}(k-i) \\
&= P_\epsilon^{2\times n_c^*}(1-P_\epsilon)^k \times \sum_{i=0}^{k} \binom{n_c^*+i-1}{i} \times \binom{n_c^*+k-i-1}{k-i} \\
&= \binom{2\times n_c^*+k-1}{k} P_\epsilon^{2\times n_c^*} \times (1-P_\epsilon)^k
\end{aligned} \tag{4.50}
$$

Having the probability distribution function (Eq. (4.50)) allows us, for the checkpoint assignment $\bar{n}_c^* = [n_c^*, n_c^*]$, to calculate $\mathbb{A}_{\bar{n}_c^*}(D)$ using the following expression:

$$
\begin{aligned}
\mathbb{A}_{\bar{n}_c^*}(D) &= \sum_{{}^{\bar{n}_c^*}t_k \le D} p'_{\bar{n}_c^*}(k) \\
&= \sum_{{}^{\bar{n}_c^*}t_k \le D} \binom{2\times n_c^*+k-1}{k} P_\epsilon^{2\times n_c^*} \times (1-P_\epsilon)^k
\end{aligned} \tag{4.51}
$$

The expression in Eq. (4.51) calculates $\mathbb{A}_{\bar{n}_c^*}(D)$ for the assumed scenario of two jobs and a global deadline. Following the same reasoning as for the scenario of two jobs, we derive expressions for a general scenario that consists of m jobs. For the general case of m jobs we derive the following expression:

$$
{}^{\bar{n}_c^*}t_k = m \cdot (T + n_c^* \times \tau) + k \times (\frac{T}{n_c^*} + \tau) \tag{4.52}
$$

$$
p'_{\bar{n}_c^*}(k) = \binom{m\times n_c^*+k-1}{k} P_\epsilon^{m\times n_c^*} \times (1-P_\epsilon)^k \tag{4.53}
$$

$$
\mathbb{A}_{\bar{n}_c}(D) = \sum_{{}^{\bar{n}_c^*}t_k \le D} \binom{m\times n_c^*+k-1}{k} P_\epsilon^{m\times n_c^*} \times (1-P_\epsilon)^k \tag{4.54}
$$

Important to note is that $\mathbb{A}_{\bar{n}_c}(D)$ is calculated while assuming a checkpoint assignment where each job is using n_c^* checkpoints, and n_c^* is obtained by performing an optimization for a single job while using the notion of local deadlines D' which are not part of the stated problem formulation.

If we study the expressions in Eq. (4.3), Eq. (4.6), and Eq. (4.8), and compare them with the expressions in Eq. (4.52), Eq. (4.53), and Eq. (4.54), respectively, we see that the latter expressions are similar with the former ones. Indeed, assuming a single job with a processing time $m \times T$, which uses $m \times n_c^*$ checkpoints transforms the expressions Eq. (4.3), Eq. (4.6), and Eq. (4.8) into the expressions we have presented in Eq. (4.52), Eq. (4.53), and Eq. (4.54), respectively. Therefore, this motivates the second approach, *i.e.* Single Large Job.

SINGLE LARGE JOB This approach suggests to treat the set of jobs as one single large job, and thus makes the problem equivalent to the problem of having a single job and a deadline (see Figure 4.4(c)). The approach considers that a set of m jobs, where each job has a processing time T, is equivalent to a single job with a processing time $m \times T$. Using the optimization framework for a single job, the approach finds the optimal number of checkpoints n_c^{**} that results in the maximal LoC $\Lambda_{n_c^{**}}(D)$. Observe that here n_c^{**} is the optimal number of checkpoints that should be used for the single job with a processing time $m \times T$. To obtain the number of checkpoints that should be used by each job it is sufficient to divide n_c^{**} with the number of jobs m. This results in a checkpoint assignment where each job uses exactly $n_c^* = \frac{n_c^{**}}{m}$ checkpoints. The advantage of this approach in comparison to the Local Optimization approach (Section 4.2.2) is that we are able to identify the number of checkpoints that should be used by each job without introducing local deadlines, which are not part of the stated problem formulation. However, there are two main disadvantages. First, the optimal number of checkpoints n_c^{**} for the single large job does not necessarily need to be a multiple of the number of jobs in the set m, and thus we cannot guarantee that by adopting a checkpoint assignment such that each job uses $n_c^* = \lfloor \frac{n_c^{**}}{m} \rfloor$ will provide the maximal $\Lambda_{\bar{n}_c}(D)$ which we aim to find. Second, this approach imposes a limit that all jobs need to use the same number of checkpoints. However, using a checkpoint assignment $\bar{n}_c$ where all jobs use the same number of checkpoints does not guarantee that the achieved $\Lambda_{\bar{n}_c}(D)$ is the maximal LoC with respect to the given global deadline. This motivates the need to investigate how to calculate $\Lambda_{\bar{n}_c}(D)$ when the number of checkpoints is individually assigned to each job in the set.

4.2.3. EXHAUSTIVE SEARCH

In the previous section, we investigated two approaches that directly apply the solution for a single job. The main drawback of these approaches is that they report an LoC which is achieved by assigning the same number of checkpoints to all jobs. However, there is no guarantee that the maximal LoC with respect to a given global deadline is achieved when all jobs use the same number of checkpoints. To obtain the maximal LoC and the optimal checkpoint assignment, in this section, we review an Exhaustive Search approach. This approach evaluates the LoC for all *valid* checkpoint assignments $\bar{n}_c = [n_{c_1}, n_{c_2} ... n_{c_m}]$, where a *valid* checkpoint assignment is a checkpoint assignment that does not violate the global deadline when no errors occur (see Eq. (4.61) and Eq. (4.62)). We denote with V the set of all *valid* checkpoint assignments $\bar{n}_c$. For each checkpoint assignment $\bar{n}_c$, the LoC with respect to the global deadline D is computed, and then the checkpoint assignment $\bar{n}_c^*$

for which the maximal LoC is obtained is reported as the optimal assignment, *i.e.*:

$$\mathbb{\Lambda}_{\bar{n}_c^*}(D) = MAX\{\mathbb{\Lambda}_{\bar{n}_c}(D) : \forall\ \bar{n}_c \in V\}$$

Since the solution from this approach is obtained after exploring all *valid* checkpoint assignments, we can guarantee its optimality. As the maximal LoC is obtained after evaluating the LoC for all *valid* checkpoint assignments, there is a need for an expression to calculate the LoC for a given checkpoint assignment. Therefore, in the following text, we derive an expression to compute the LoC for a given checkpoint assignment. To derive the expression, first, we start with a scenario which consists of two jobs, and then generalize the expression for a scenario which consists of m jobs.

Let us assume a scenario which consists of two jobs, each with a processing time T, and a global deadline D. Further, let us assume that n_{c_1} checkpoints are assigned to the first job, and n_{c_2} checkpoints are assigned to the second job, *i.e.* a checkpoint assignment $\bar{n}_c = [n_{c_1}, n_{c_2}]$. Each of the jobs is expected to complete only at equidistant discrete time moments, that can be expressed by a discrete variable as presented in Eq. (4.3). However, the discrete time moments when both jobs complete are no longer equidistant. For the given scenario, the expected time for both jobs to complete can be computed with the following expression:

$${}^{n_{c_{1,2}}}t_{k_{1,2}} = T + n_{c_1}\tau + T + n_{c_2}\tau + k_1(\frac{T}{n_{c_1}} + \tau) + k_2(\frac{T}{n_{c_2}} + \tau) \tag{4.55}$$

Eq. (4.55) includes: the processing time for the first job along with the checkpointing overhead of taking n_{c_1} checkpoints, the processing time of the second job along with the checkpointing overhead of taking n_{c_2} checkpoints, the penalty of having k_1 erroneous execution segments during the execution of the first job, and the penalty of having k_2 erroneous execution segments during the execution of the second job.

In Figure 4.5 we illustrate the completion time ${}^{n_{c_{1,2}}}t_{k_{1,2}}$ for two jobs. The first job uses four checkpoints, and the second job uses three checkpoints. In the best case, when no errors occur, both jobs complete after executing four execution segments for the first job and three execution segments for the second job. This is illustrated with $t_{0,0}$ (see Figure 4.5). However, in the case that errors occur during the execution of each of the jobs, to handle the errors each job needs to re-execute some execution segments. This affects the completion time. For example, $t_{1,1}$ in Figure 4.5 represents the completion time when both jobs have re-executed (due to errors) one execution segment.

To compute the LoC with respect to a given global deadline D, we first calculate the probability $p_{n_{c_{1,2}}}(k_{1,2})$ that both jobs complete at time ${}^{n_{c_{1,2}}}t_{k_{1,2}}$. Since both jobs are independent processes, the probability that both jobs complete

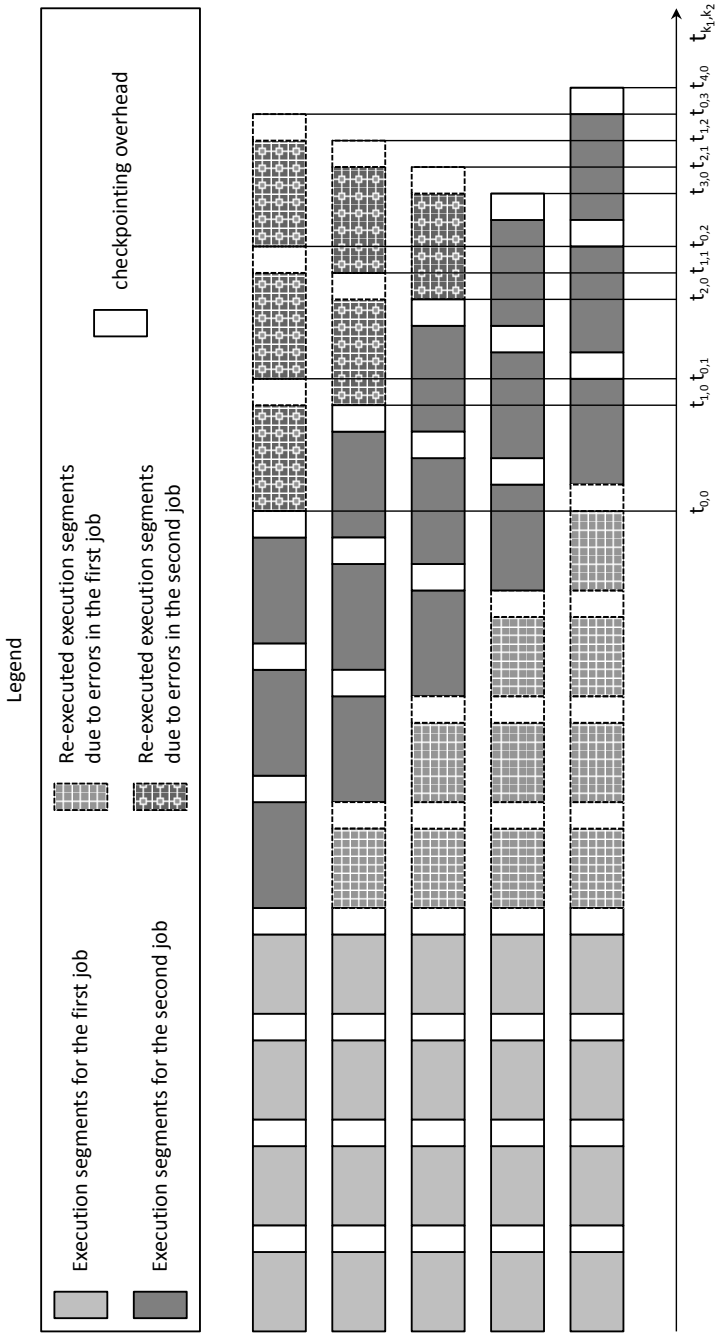


Figure 4.5.: Completion time

at time ${}^{n_{c_{1,2}}}t_{k_{1,2}}$ is the combined probability that the first job has completed at time t_{k_1}, and the second job has completed at time t_{k_2}. This is given with the following expression:

$$p_{n_{c_{1,2}}}(k_{1,2}) = p_{n_{c_1}}(k_1) \times p_{n_{c_2}}(k_2) \tag{4.56}$$

Summing the terms of the probability distribution function, presented in Eq. (4.56), for all ${}^{n_{c_{1,2}}}t_{k_{1,2}} \leq D$ provides the LoC with respect to the given global deadline D. Eq. (4.57) provides the LoC with respect to the global deadline when the checkpoint assignment $\bar{n}_c = [n_{c_1}, n_{c_2}]$ is used.

$$\begin{aligned} \mathbb{\Lambda}_{\bar{n}_c}(D) &= \sum_{{}^{n_{c_{1,2}}}t_{k_{1,2}} \leq D} p_{n_{c_{1,2}}}(k_{1,2}) \\ &= \sum_{{}^{n_{c_{1,2}}}t_{k_{1,2}} \leq D} p_{n_{c_1}}(k_1) \times p_{n_{c_2}}(k_2) \end{aligned} \tag{4.57}$$

So far, we showed how to calculate the LoC with respect to a given global deadline for a scenario of two jobs, where a number of checkpoints is assigned to each job. Following the same reasoning, we show next how to calculate the LoC for a scenario of m jobs, while assuming that a checkpoint assignment $\bar{n}_c$, *i.e.* the number of checkpoints used by each job, is given.

For the general case, *i.e.* a set of m jobs, where for each job i the processing time is T and the number of checkpoints is n_{c_i}, the expected time for all jobs to complete is expressed with the following expression:

$${}^{\bar{n}_c}t_{\bar{k}} = m \times T + \sum_{i=1}^{m} n_{c_i}\tau + \sum_{i=1}^{m} k_i\left(\frac{T}{n_{c_i}} + \tau\right) \tag{4.58}$$

In Eq. (4.58), the index $\bar{k}$ represents a vector, *i.e.* $\bar{k} = [k_1, k_2 ... k_m]$, where each element k_i represents the number of erroneous execution segments during the execution of job i.

For a given checkpoint assignment $\bar{n}_c = [n_{c_1}, n_{c_2} ... n_{c_m}]$, where n_{c_i} denotes the number of checkpoints assigned to job i, the probability that all jobs complete at a given time ${}^{\bar{n}_c}t_{\bar{k}}$ is evaluated with the following probability distribution function:

$$p_{\bar{n}_c}(\bar{k}) = \prod_{i=1}^{m} p_{n_{c_i}}(k_i) \tag{4.59}$$

Finally, the LoC with respect to a given global deadline D, for a given checkpoint assignment, is computed with the following expression:

$$\mathbb{\Lambda}_{\bar{n}_c}(D) = \sum_{{}^{\bar{n}_c}t_{\bar{k}} \leq D} p_{\bar{n}_c}(\bar{k}) = \sum_{{}^{\bar{n}_c}t_{\bar{k}} \leq D} \prod_{i=1}^{m} p_{n_{c_i}}(k_i) \tag{4.60}$$

Using Eq. (4.60), the Exhaustive Search approach obtains the maximal LoC by computing the LoC for all *valid* checkpoint assignments. A checkpoint assignment $\bar{n}_c$ is *valid* if and only if the following two conditions are satisfied:

$$n_{c_i} \geq 1, \forall i \in [1, m] \tag{4.61}$$

$$\sum_{i=0}^{m} n_{c_i} \leq \lfloor \frac{D - m \cdot T}{\tau} \rfloor \tag{4.62}$$

The condition in Eq. (4.61) expresses the lower bound for the number of checkpoints to be used per job. Indeed, when RRC is applied, a job needs to use at least one checkpoint (after its execution) to verify that it has completed without any errors. The condition in Eq. (4.62) is related to the upper bound of the total number of checkpoints, *i.e.* the sum of the number of checkpoints used by each job. As we have presented in the expression for the expected completion time (see Eq. (4.58)), even in the best case scenario when no errors occur, *i.e.* ${}^{\bar{n}_c}t_{\bar{0}}$ where $\bar{0}$ represents a null vector, the expected completion time includes the processing time for each job along with the checkpointing overhead per job due to taking n_{c_i} checkpoints which is presented with the following expression:

$${}^{\bar{n}_c}t_{\bar{0}} = \sum_{i=0}^{m}(T + n_{c_i} \cdot \tau) = m \cdot T + \tau \cdot \sum_{i=0}^{m} n_{c_i} \tag{4.63}$$

Observe, from Eq. (4.63), that ${}^{\bar{n}_c}t_{\bar{0}}$ depends on the total number of checkpoints. If the total number of checkpoints is very large, it may happen that the global deadline is violated even in the best case scenario, *i.e* ${}^{\bar{n}_c}t_{\bar{0}} > D$. In such case, the LoC with respect to the given global deadline is equal to zero. Thus, we only need to investigate checkpoint assignments $\bar{n}_c$ for which the best case expected completion time ${}^{\bar{n}_c}t_{\bar{0}}$ does not violate the global deadline D, *i.e.* ${}^{\bar{n}_c}t_{\bar{0}} \leq D$, which holds if the condition in Eq. (4.61) is satisfied.

Since this approach calculates the LoC of meeting the global deadline for all *valid* checkpoint assignments, we can guarantee that this approach always finds the maximal LoC and the optimal checkpoint assignment. However, the main drawback of this approach is that it is computationally intensive, and extremely time-consuming (this is shown in Section 4.2.5). The reason is that the number of *valid* checkpoint assignments $\bar{n}_c$ grows rapidly as the number of jobs m increases. Therefore, in the following section, we propose a search method (heuristic) that aims to maximize the LoC at a significantly lower computational time.

4.2.4. SEMI-EXHAUSTIVE SEARCH

In this section, we propose a search method, *i.e.* Semi-Exhaustive Search, that substantially reduces the time required to obtain the optimal checkpoint

assignment and the maximal LoC in comparison to the Exhaustive Search approach described in the previous section. For convenience, we divide the section into two subsections. First, we provide the motivation for this method, and then we discuss the method.

MOTIVATION The Semi-Exhaustive Search method is based on the idea that even though the search space, *i.e.* set of all *valid* checkpoint assignments, grows rapidly as the number of jobs in the set increases, there is no need to explore all *valid* checkpoint assignments. There are two reasons for this, expressed with the following statements:

- *Statement* I: Some checkpoint assignments are equivalent;
- *Statement* II: Some checkpoint assignments do not contribute in finding the maximal LoC.

For *Statement* I, we first need to define the notion of equivalent checkpoint assignments. For a given checkpoint assignment $\bar{n}_c = [n_{c_1}, n_{c_2}, n_{c_3} ... n_{c_m}]$, we denote each permutation of $\bar{n}_c$ as an equivalent checkpoint assignment to $\bar{n}_c$. Therefore, computing the LoC for a checkpoint assignment $\bar{n}_c$ is sufficient, and there is no need to compute the LoC for any other checkpoint assignment which is a permutation of $\bar{n}_c$. Observe that the notion of equivalent checkpoint assignments comes from the fact that in the stated problem formulation we consider a set of jobs, where all jobs have the same processing time T. Since all jobs have the same processing time, it is irrelevant how the number of checkpoints are assigned to different jobs, and next we show that this statement is valid. To justify this claim, let us examine carefully the expressions for calculating the expected completion time and the LoC. The expected completion time ${}^{\bar{n}_c}t_{\bar{k}}$ for a checkpoint assignment $\bar{n}_c$ is calculated with Eq. (4.58). The index $\bar{k}$ represents a vector $[k_1, k_2, ... k_m]$ where each element k_i denotes the number of erroneous execution segments (the number of execution segments that require re-execution) for each job i, and contributes in the expected completion time with a factor $k_i(\frac{T}{n_{c_i}} + \tau)$. When an equivalent checkpoint assignment $\bar{n}_c^{\dagger}$ (a permutation of $\bar{n}_c$) is used, there exists a vector $\bar{k}^{\dagger}$ which itself is a permutation of $\bar{k}$ such that ${}^{\bar{n}_c}t_{\bar{k}} = {}^{\bar{n}_c^{\dagger}}t_{\bar{k}^{\dagger}}$. The former equality comes from the fact that for an arbitrary element n_{c_i} in $\bar{n}_c$ there is an element $n_{c_j}^{\dagger}$ in $\bar{n}_c^{\dagger}$ such that $n_{c_i} = n_{c_j}^{\dagger}$, and further $k_i = k_j^{\dagger}$. Thus, the set of expected completion times $\{{}^{\bar{n}_c}t_{\bar{k}} : {}^{\bar{n}_c}t_{\bar{k}} \leq D\}$ when a checkpoint assignment $\bar{n}_c$ is used is equivalent with the set $\left\{{}^{\bar{n}_c^{\dagger}}t_{\bar{k}^{\dagger}} : {}^{\bar{n}_c^{\dagger}}t_{\bar{k}^{\dagger}} \leq D\right\}$. Since the LoC is calculated as a sum of terms from the probability distribution function $p_{\bar{n}_c}(\bar{k})$, equal number of terms are summed for both equivalent checkpoint assignments. Observing the expression for $p_{\bar{n}_c}(\bar{k})$ given in Eq. (4.59), one notes that due to the commutative

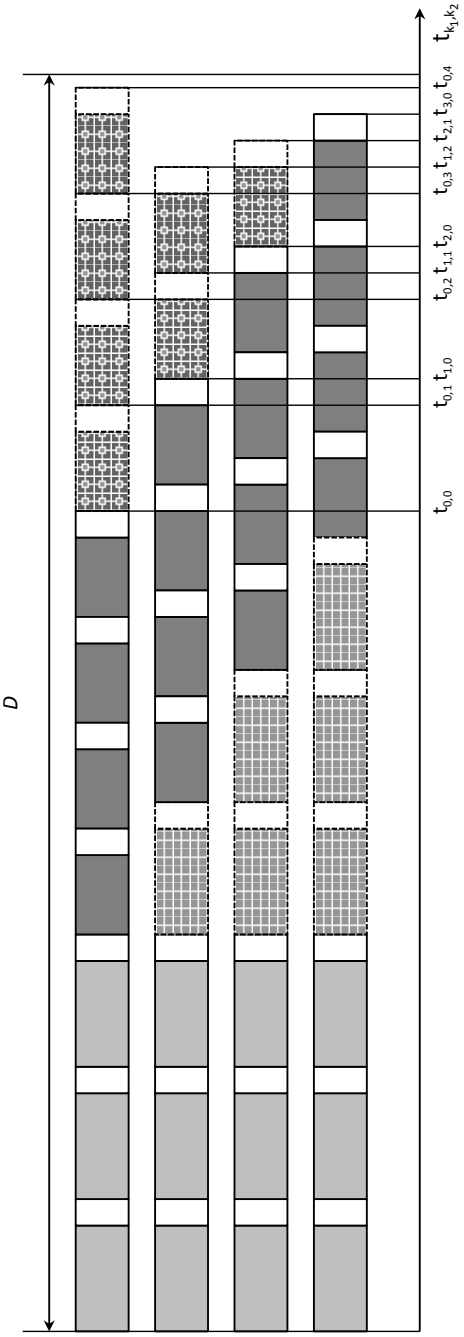
property of multiplication the following equality holds $p_{\bar{n}_c}(\bar{k}) = p_{\bar{n}_c^\dagger}(\bar{k}^\dagger)$. Finally, since for two equivalent checkpoint assignments $\bar{n}_c$ and $\bar{n}_c^\dagger$ the same number of terms are used to calculate the LoC, and since all these terms are equal for both checkpoint assignments, we conclude that $\mathbb{A}_{\bar{n}_c}(D) = \mathbb{A}_{\bar{n}_c^\dagger}(D)$ which justifies to abandon redundant computation of the LoC for equivalent checkpoint assignments.

We demonstrate that the same LoC is obtained when equivalent checkpoint assignment are used with the following example. Consider the following scenario: given is a set of two jobs, where each of the jobs has a processing time $T = 1200$ t.u., a global deadline $D = 4000$ t.u., a checkpointing overhead $\tau = 20$ t.u., and two checkpoint assignments $\bar{n}_c^\dagger = [3,4]$ and $\bar{n}_c^\ddagger = [4,3]$. Observe that according to the definition of equivalent checkpoint assignments, $\bar{n}_c^\dagger = [3,4]$ and $\bar{n}_c^\ddagger = [4,3]$ are equivalent due to the fact that $\bar{n}_c^\dagger$ is a permutation of $\bar{n}_c^\ddagger$, and vice-versa. In Figure 4.6 and Figure 4.7, we depict the expected completion time for $\bar{n}_c^\dagger$ and $\bar{n}_c^\ddagger$, respectively. Both Figure 4.6 and Figure 4.7 use the same legend as depicted in Figure 4.5. As shown in Figure 4.6 and Figure 4.7, in both cases for $\bar{n}_c^\dagger$ and $\bar{n}_c^\ddagger$, the sets of discrete time moments when both jobs complete are equivalent, *e.g.* $t_{0,1}$ for $\bar{n}_c^\dagger$ is the same as $t_{1,0}$ for $\bar{n}_c^\ddagger$ etc. Further, calculating the probability that both jobs complete at a given discrete time moment, by using the expression presented in Eq. (4.59), shows that $p_{\bar{n}_c^\dagger}(k_1,k_2)$ is equal to $p_{\bar{n}_c^\ddagger}(k_2,k_1)$. Since for both cases, *i.e.* when using checkpoint assignments $\bar{n}_c^\dagger$ and $\bar{n}_c^\ddagger$, the sets of discrete time moments when both jobs complete are equivalent and the probabilities $p_{\bar{n}_c}(t_{\bar{k}})$ are equal, leads to the fact the LoC of meeting the global deadline D is the same when using $\bar{n}_c^\dagger$ and $\bar{n}_c^\ddagger$. Therefore, this justifies to discard redundant computation of the LoC for equivalent checkpoint assignments.

To avoid exploration of equivalent checkpoint assignments, it is required to only explore such checkpoint assignments for which the following expression holds:

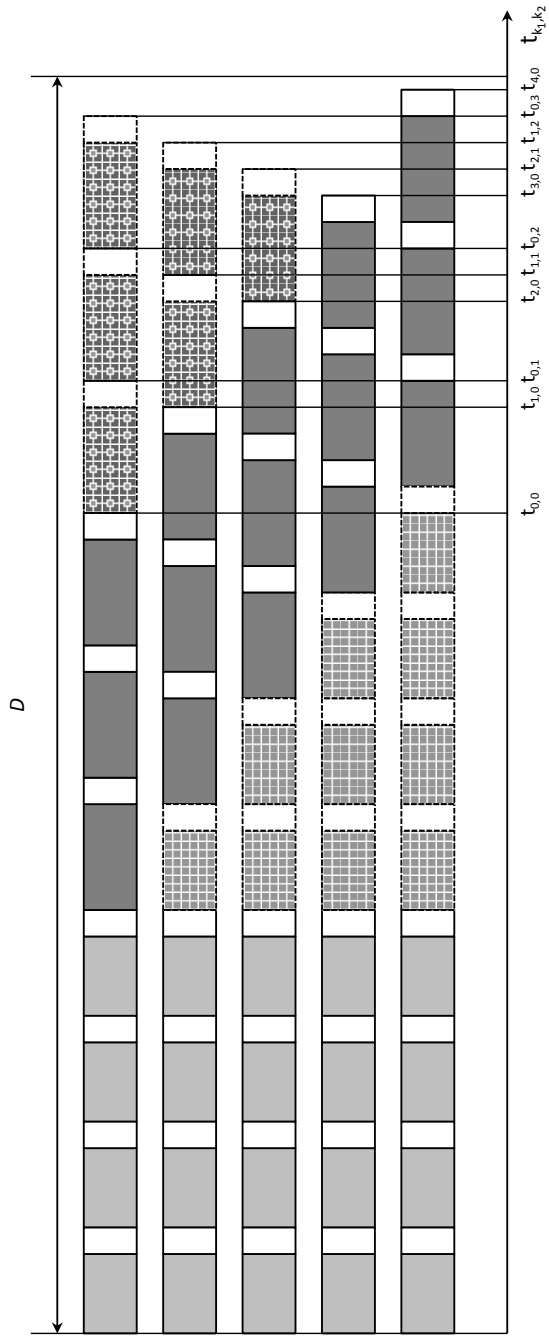
$$n_{c_1} \leq n_{c_2} \leq ... \leq n_{c_i} \leq ... \leq n_{c_m} \tag{4.64}$$

To justify *Statement* II, *i.e.* some checkpoint assignments do not contribute in finding the maximal LoC, we first provide some information on how we explore the search space. We explore the search space by first removing all *valid* equivalent checkpoint assignments, *i.e.* only checkpoint assignments that satisfy the property in Eq. (4.64) are explored. Next, the checkpoint assignments $\bar{n}_c$ are ordered in an ascending order, and then iteratively are explored. This ordering is done by consecutively sorting the checkpoint assignments in an ascending order based on n_{c_i} where i varies from 1 to m. Hence, once all *valid* checkpoint assignments are sorted in an ascending order based on n_{c_1}, the checkpoint assignments are next sorted in an ascending order based on



Note*: Observe the legend depicted in Figure 4.5

Figure 4.6: Completion time for checkpoint assignment $\bar{n}_c^{\dagger} = [3, 4]$



Note*: Observe the legend depicted in Figure 4.5

Figure 4.7.: Completion time for checkpoint assignment $\bar{n}_c^\dagger = [4, 3]$

n_{c_2} (while preserving the ascending order of n_{c_1}), and this process is repeated. For such ordering of the checkpoint assignment, after exploring a checkpoint assignment $\bar{n}_c$, there exist a unique checkpoint assignment that follows. We use the following function to obtain the next checkpoint assignment to be explored:

$$\begin{aligned} \bar{n}_c^N &= [n_{c_1}, n_{c_2}, ...n_{c_i}+1, n_{c_i}+1, ...n_{c_i}+1] \\ next(\bar{n}_c, i) &= \begin{cases} \bar{n}_c^N, \text{ iff } \bar{n}_c^N \text{ is } \textit{valid} \text{ and } i \geq 1 \\ next(\bar{n}_c, i-1), \text{ iff } \bar{n}_c^N \text{ is not } \textit{valid} \text{ and } i \geq 1 \\ \emptyset, \text{ iff } i = 0 \end{cases} \end{aligned}$$

The function $next(\bar{n}_c, i)$ takes two arguments: a checkpoint assignment $\bar{n}_c$ and an *update* index i. First, the function calculates $\bar{n}_c^N$ for the given $\bar{n}_c$ by incrementing the number of checkpoints at index i, *i.e.* $n_{c_i} \rightarrow n_{c_i}+1$, and assigning $n_{c_i}+1$ to all n_{c_j} where $j \in [i+1, m]$. If $\bar{n}_c^N$ is a *valid* checkpoint assignment, then $\bar{n}_c^N$ is the next checkpoint assignment to be explored. However, if $\bar{n}_c^N$ is not a *valid* checkpoint assignment, then the function steps into a recursive call with the same checkpoint assignment $\bar{n}_c$ and an *update* index $i-1$, *i.e.* $next(\bar{n}_c, i-1)$. If the function is invoked with an *update* index $i = 0$, the function returns an empty set, which means that $\bar{n}_c$ is the last *valid* checkpoint assignment in the search space.

To obtain the next checkpoint assignment for a given $\bar{n}_c$, $next(\bar{n}_c, i)$ is always invoked with the following arguments: the current checkpoint assignment $\bar{n}_c$ and the number of jobs in the set m, *i.e.* $next(\bar{n}_c, m)$.

We demonstrate how to obtain the next checkpoint assignment for a given $\bar{n}_c$ with an example. Consider a set of three jobs, a checkpoint assignment $\bar{n}_c = [2, 3, 4]$, and assume that the total number of checkpoints must be lower or equal to 10 (see the condition in Eq. (4.62)). To obtain the next checkpoint assignment, we use $next(\bar{n}_c, 3)$. Observe that the function is invoked with the current checkpoint assignment $\bar{n}_c = [2, 3, 4]$ and an *update* index 3, which is equal to the number of jobs in the set. The function first calculates $\bar{n}_c^N = [2, 3, 5]$, which is a *valid* checkpoint assignment, *i.e.* the total number of checkpoints, $2+3+5$, is lower or equal to 10. Therefore, the next checkpoint assignment that is explored is $\bar{n}_c = [2, 3, 5]$. After exploring $\bar{n}_c = [2, 3, 5]$, assume that we want to explore the next checkpoint assignment. Therefore, we again invoke the function $next(\bar{n}_c, 3)$, where $\bar{n}_c = [2, 3, 5]$. The function first calculates $\bar{n}_c^N = [2, 3, 6]$ which is not a *valid* checkpoint assignment due to the fact that the total number of checkpoints is larger than 10. Therefore, the function steps in a recursive call, but this time with different arguments, *i.e.* $next(\bar{n}_c, 2)$. According to its definition, the function calculates a new value

for $\bar{n}_c^N$, *i.e.* $\bar{n}_c^N = [2,4,4]$, which is a *valid* checkpoint assignment, and thus the next checkpoint assignment to be explored. Hence, the checkpointing assignment which is explored after $\bar{n}_c = [2,3,5]$ is $\bar{n}_c = [2,4,4]$. Let us assume that the current checkpoint assignment is $\bar{n}_c = [3,3,4]$ and we want to find the next one by using $next(\bar{n}_c, 3)$. The function first calculates $\bar{n}_c^N = [3,3,5]$, which is not a *valid* checkpoint assignment as the total number of checkpoints is larger than 10, and therefore the function steps in a recursive call, *i.e.* $next(\bar{n}_c, 2)$. After this recursive call, the new $\bar{n}_c^N = [3,4,4]$ is calculated, and again this is not a *valid* checkpoint assignment, which forces the function to step in another recursive call, $next(\bar{n}_c, 1)$. At this recursive call $\bar{n}_c^N = [4,4,4]$ is calculated, and again this is not a *valid* checkpoint assignment which forces the function to step in yet another recursive call $next(\bar{n}_c, 0)$. Since in the last recursive call the function was invoked with an *update* index $i = 0$, the function returns an empty set, which means that the checkpoint assignment $\bar{n}_c = [3,3,4]$ is the last *valid* checkpoint assignment in the search space.

As the search space is traversed in an iterative manner, we observe a set of checkpoint assignments that only differ in the last index position. We introduce the term *sequence of checkpoint assignments* $\{\bar{n}_c\}$ to denote the set of such checkpoint assignments. Two checkpoint assignments $\bar{n}_c^\dagger$ and $\bar{n}_c^\ddagger$ belong to the same sequence of checkpoint assignments $\{\bar{n}_c\}$, if and only if $n_{c_i}^\dagger = n_{c_i}^\ddagger, \forall i \in [1, m-1]$ and $n_{c_m}^\dagger \neq n_{c_m}^\ddagger$. Thus, a sequence of checkpoint assignments $\{\bar{n}_c\}$ consists of a set of checkpoint assignments $\{[n_{c_1}, n_{c_2} ... n_{c_{m-1}}, X]\}$ where $X \in [n_{c_{m-1}}, n_{c_{max}}]$. Observe that we can always compute $n_{c_{max}}$ by using Eq. (4.62).

To summarize the discussion on the exploration of the search space, we state that the search space is traversed by exploring consecutive sequences of checkpoint assignments. Next, we show that the search space can be efficiently reduced by (1) not exploring all checkpoint assignments that belong to a sequence of checkpoint assignments, and (2) discarding exploration of entire sequences of checkpoint assignments. The reason for this is that all these checkpoint assignments that are discarded do not contribute in finding the maximal LoC.

To show that there is no need to explore all checkpoint assignments that belong to a sequence of checkpoint assignments, let us now focus on the exploration of a given sequence of checkpoint assignments $\{\bar{n}_c\}$. The given $\{\bar{n}_c\}$ consists of checkpoint assignments where the number of checkpoints for the first $m-1$ jobs in the set is fixed, and only the number of checkpoints for the last job varies. Thus, exploring a single sequence of checkpoint assignments $\{\bar{n}_c\}$ becomes equivalent as exploring the number of checkpoints to be used for a single job (only the number of checkpoints for the last job varies). As we have already presented for the the case of a single job, in-

creasing the number of checkpoints up to a certain value (breaking point) improves the LoC. However, increasing the number of checkpoints beyond this value degrades the LoC and may even lead to a zero LoC. The reason is that by increasing the number of checkpoints beyond the breaking point, the checkpointing overhead increases and limits the number of re-executions that can take place without violating the deadline, which implicitly reduces the number of terms from the probability distribution function that are used to compute the LoC, resulting in a lower LoC. In the context of multiple jobs, when exploring checkpoint assignments $\bar{n}_c$ which belong to a given sequence of checkpoint assignments $\{\bar{n}_c\}$, the number of checkpoints for all jobs except for the last one is fixed, and only the number of checkpoints for the last job varies. The number of checkpoints for the last job has to be larger than or equal to $n_{c_{m-1}}$ to avoid exploration of equivalent checkpoint assignments (see Eq. (4.64)). When exploring a single $\{\bar{n}_c\}$, the number of checkpoints for the last job varies within the range $[n_{c_{m-1}}, n_{c_{max}}]$. As consecutive checkpoint assignments from the sequence are explored the number of checkpoints for the last job increases, starting from $n_{c_{m-1}}$. By increasing the number of checkpoints used for the last job, the execution segments for the last jobs become shorter. This allows more re-executions of execution segments (only execution segments from the last job) to take place without violating the deadline at the cost of increasing the checkpointing overhead due to the higher number of checkpoints. However, increasing the number of checkpoints for the last job, and therefore increasing the checkpointing overhead is the reason to limit the number of re-executions of execution segments from other jobs. To clarify this discussion, assume three jobs, each having the processing time $T = 1000$ t.u., a global deadline $D = 3600$ t.u., and assume a checkpointing overhead $\tau = 10$ t.u. Further, assume we explore the sequence of checkpoint assignments $\{\bar{n}_c\} = [2, 2, X]$ where $X \in [2, 56]$ (the upper bound is computed according to Eq. (4.62)). When $X = 2$, only one execution segment from any of the three jobs may be re-executed without violating the global deadline. Increasing the number of checkpoints for the last job to $X = 4$ improves the LoC with respect to the global deadline due to the fact that the deadline is met even if one of the execution segments from the first or the second job has to be re-executed, or even if two execution segments from the last job have to be re-executed. Increasing X beyond 7 would allow more execution segments from the last job to be re-executed at the cost that no re-executions of any execution segments from the first two jobs would be tolerated. For example when $X = 8$, the deadline will be met even if three execution segments from the last job are affected by errors. However, for $X = 8$, the deadline will be violated if a single execution segment in the first or the second job is affected by errors which in turn degrades the LoC.

From this discussion we conclude that for a given $\{\bar{n}_c\}$, there exist a checkpoint assignment $\bar{n}_c = [n_{c_1}, n_{c_2}, n_{c_3}...n^*_{c_m}]$ that provides the locally optimal LoC. Since exploring all the checkpoint assignments which belong to $\{\bar{n}_c\}$ is the same as varying the number of checkpoints for the last job in the range $[n_{c_{m-1}}, n_{c_{max}}]$, we can guarantee that $n^*_{c_m} \in [n_{c_{m-1}}, n_{c_{max}}]$. Important to observe here is that as the number of checkpoints for the last job increases from $n_{c_{m-1}}$ to $n^*_{c_m}$ the LoC constantly increases and reaches the highest value (highest LoC among all checkpoint assignments in the given sequence) at $n^*_{c_m}$. However, increasing the number of checkpoints beyond $n^*_{c_m}$, *i.e.* when the number of checkpoints for the last job increases from $n^*_{c_m} + 1$ to $n_{c_{max}}$, results in a lower LoC.

As the goal is to obtain the maximal (globally optimal) LoC, only checkpoint assignments that are local optima can be candidates to reach this goal. Therefore, for each sequence of checkpoint assignments $\{\bar{n}_c\}$, we do not need to explore the entire sequence. Instead, we only need to explore the checkpoint assignments $\{[n_{c_1}, n_{c_2}...n_{c_{m-1}}, X]\}$, where $X \in [n_{c_{m-1}}, n^*_{c_m} + 1]$ and the local optimum is achieved for $n^*_{c_m}$. The rest of the checkpoint assignments from $\{\bar{n}_c\}$, *i.e.* $\{[n_{c_1}, n_{c_2}...n_{c_{m-1}}, X]\}$, where $X \in [n^*_{c_m} + 2, n_{c_{max}}]$, do not contribute in finding the maximal LoC and evaluating the LoC for these checkpoint assignments only deviates from the maximal LoC.

Previously we introduced the function $next(\bar{n}_c, i)$ to obtain the next checkpoint assignment to be explored. To obtain the next checkpoint assignment the function is invoked with the following arguments: $\bar{n}_c$ the current checkpoint assignment and m the number of jobs in the set, *i.e.* $next(\bar{n}_c, m)$. The currently explored checkpoint assignment $\bar{n}_c$ belongs to a sequence of checkpoint assignments $\{\bar{n}_c\}$. To avoid exploring the rest of the checkpoint assignments that belong to the given sequence of checkpoint assignments, we use the same function $next(\bar{n}_c, i)$, but this time we invoke it with the following arguments: $\bar{n}_c$ and $m - 1$. By invoking $next(\bar{n}_c, m - 1)$, the function returns the first checkpoint assignment of the successive sequence of checkpoint assignments, thus skipping the rest of the checkpoint assignments that belong to the current sequence of checkpoint assignments $\{\bar{n}_c\}$.

So far, we showed that the search space can be reduced by not exploring all checkpoint assignments that belong to a sequence of checkpoint assignments. However, as we mentioned earlier, further reduction of the search space is possible by discarding exploration of entire sequences of checkpoint assignments, and next we elaborate on this. The reason to discard exploration of entire sequences of checkpoint assignments is that the total checkpointing overhead after exploring consecutive sequences of checkpoint assignments becomes dominant and limits the amount of re-executions of execution segments of all the jobs in the set. As the search space is explored in an iterative manner, it means that consecutive sequences of checkpoint assignments are

explored. For consecutive sequences of checkpoint assignments the number of checkpoints per job increases. That means the execution segments of all jobs become shorter, which makes it possible to tolerate more re-executions of execution segments without violating the global deadline. That improves the LoC. However, at a certain point the total number of checkpoints, *i.e.* the sum of the number of checkpoints used per job, becomes too high incurring a very high checkpointing overhead which limits the amount of possible re-executions despite the fact that the execution segments are shorter. Out of this discussion, we conclude that the LoC with respect to the global deadline increases as consecutive sequences of checkpoint assignments are explored. However, after a given sequence of checkpoint assignments, due to the excessive checkpointing overhead, the LoC starts decreasing. This justifies to discard further exploration of consecutive sequences of checkpoint assignments. Now that we have motivated how to effectively reduce the search space, we detail our proposed search method, *i.e.* Semi-Exhaustive Search, that aims to find the optimal checkpoint assignment which results in the maximal LoC.

METHOD The method consists of two major loops: one inner and one outer loop. The inner loop iterates through checkpoint assignments that belong to the same sequence of checkpoint assignments, and it is used to obtain the locally optimal LoC. The outer loop iterates through consecutive sequences of checkpoint assignments, and it is used to obtain the globally optimal (the maximal) LoC. The block diagram of our proposed method is illustrated in Figure 4.8. As shown in Figure 4.8, the method starts by exploring the most pessimistic checkpoint assignment where the number of checkpoints that are used for each job is set to one, *i.e.* $\bar{n}_c = [1, 1, 1...1]$. Since no checkpoint assignments are explored yet, both the global optimum Λ^g and the local optimum Λ^l are set to zero. Next, we compute $\mathbb{\Lambda}_{\bar{n}_c}(D)$, by using the expression presented in Eq. (4.60), for the current checkpoint assignment $\bar{n}_c$. After this step is taken, we compare the recently computed $\mathbb{\Lambda}_{\bar{n}_c}(D)$ against Λ^l. If for the current checkpoint assignment $\bar{n}_c$ we get that $\mathbb{\Lambda}_{\bar{n}_c}(D)$ is higher than Λ^l, it means that we have found a better candidate that provides higher LoC than the currently evaluated Λ^l. Therefore, we set $\bar{n}_c^l = \bar{n}_c$, $\Lambda^l = \mathbb{\Lambda}_{\bar{n}_c}(D)$ and continue by exploring the next checkpoint assignment that belongs to the sequence $\{\bar{n}_c\}$, *i.e.* we set $\bar{n}_c = next(\bar{n}_c, m)$. However, if the comparison $\mathbb{\Lambda}_{\bar{n}_c}(D) \geq \Lambda^l$ is evaluated as false, it means that we have reached a checkpoint assignment within the sequence $\{\bar{n}_c\}$ that does not contribute in finding the global optimum. Observe that in such case the current value of Λ^l already keeps the local optimum for the current sequence $\{\bar{n}_c\}$, and this optimum is obtained for the checkpoint assignment $\bar{n}_c^l$. Since the local optimum for $\{\bar{n}_c\}$ is obtained, there is no need to continue with exploration of checkpoint assignments that belong to the

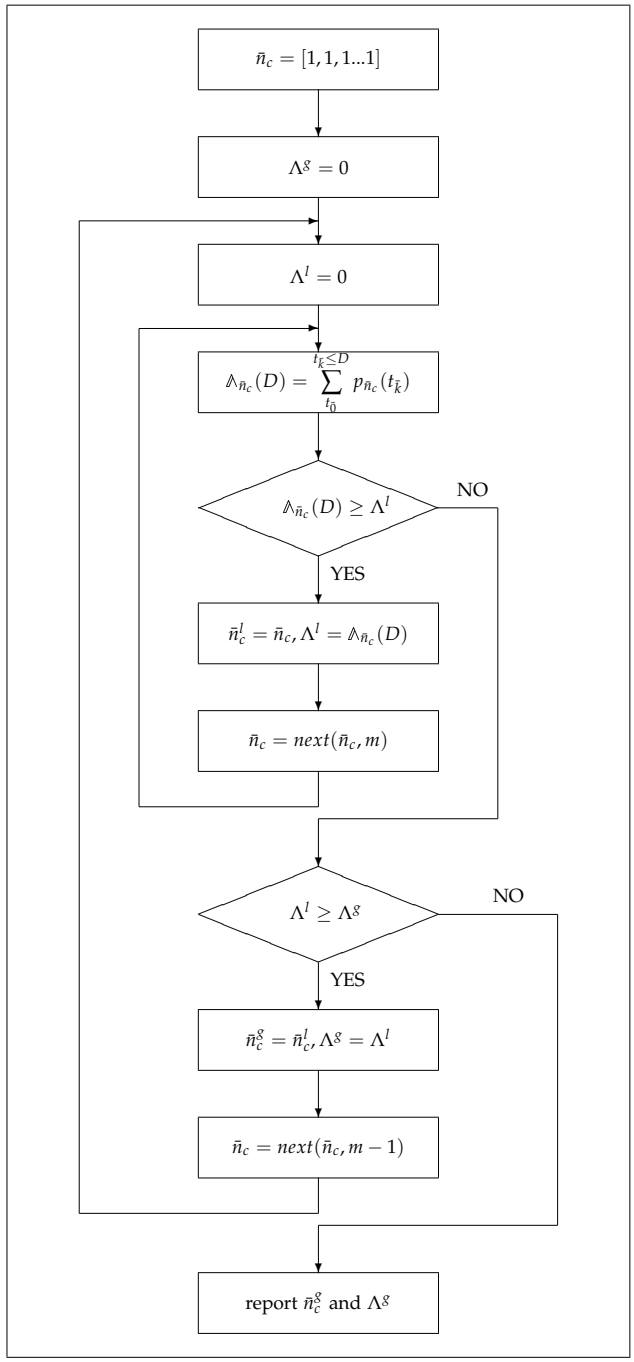


Figure 4.8.: Flow chart of the Semi-Exhaustive Search method

same sequence $\{\bar{n}_c\}$. Instead, the recently computed Λ^l is compared against Λ^g. If $\Lambda^l \geq \Lambda^g$, we have obtained a checkpoint assignment that can provide a higher LoC than the currently evaluated Λ^g. Therefore, we set $\bar{n}_c^g = \bar{n}_c^l$, $\Lambda^g = \Lambda^l$ and we continue by exploring checkpoint assignments that belong to the successive sequence of checkpoint assignments. We reach the successive sequence of checkpoint assignments by setting $\bar{n}_c = next(\bar{n}_c, m-1)$. The recently evaluated $\bar{n}_c$ belongs to a new sequence of checkpoint assignments that have not yet been explored. Therefore, the variable Λ^l, which keeps the value of the local optimum for the current sequence $\{\bar{n}_c\}$, is set to zero. However, if the comparison $\Lambda^l \geq \Lambda^g$ is evaluated as false, it means the method has reached a sequence of checkpoint assignments for which the local optimum fails to exceed the currently computed Λ^g. According to the discussion above, regarding the reduction of the search space by discarding exploration of entire sequences of checkpoint assignments, we assume that if a sequence of checkpoint assignments whose local optimum fails to exceed the currently computed Λ^g is reached, then all consecutive sequences will also fail to provide higher LoC than Λ^g. Therefore, if the comparison $\Lambda^l \geq \Lambda^g$ is evaluated as false, the method terminates by reporting Λ^g to be the maximal LoC of meeting the global deadline, and this maximum is reached by using the optimal checkpoint assignment $\bar{n}_c^g$.

Important to note is that the presented method is a heuristic to search for the optimal checkpoint assignment that results in the maximal LoC. However, we cannot guarantee that this method will always be able to find the optimal checkpoint assignment. Still, in Section 4.2.5 we demonstrate that in all experiments that we have conducted, the presented method always reports the same checkpoint assignment as the Exhaustive Search method, *i.e.* the Semi-Exhaustive Search for all conducted experiments reports the optimal solution.

4.2.5. EXPERIMENTAL RESULTS

The purpose of the experiments presented in this section is as follows. First, we show that both approaches which directly apply the solution for a single job, *i.e.* Local Optimization and Single Large Job, in the general case, do not obtain the maximal LoC for a set of jobs. Second, we show that the proposed heuristic, *i.e.* Semi-Exhaustive Search, is able to find, in most cases, the optimal checkpoint assignment and the maximal LoC. Third, we show that the Semi-Exhaustive Search is able to find the optimal checkpoint assignment and the maximal LoC in much shorter computation time when compared against the Exhaustive Search approach.

We present results for the following two experiments:

- P1: we compare the LoC with respect to the global deadline, $\mathbb{A}_{\bar{n}_c^*}(D)$, obtained from the four approaches: Local Optimization, Single Large

Job, Exhaustive Search, and Semi-Exhaustive Search

- P2: we compare the computation time for the Exhaustive Search approach against the proposed Semi-Exhaustive Search method

For the experiments, we use the input scenarios presented in Table 4.4. Each scenario is defined with the following parameters:

- m, the number of jobs in the set;
- T, the processing time for each job;
- τ, the checkpointing overhead;
- D, the global deadline; and
- P_T, the probability that no errors occur in a processor during a period of time T.

Scenario	m	T	τ	D	P_T
A	2	1000	10	2800	0.99999
B	2	1000	10	2600	0.99999
C	3	1000	10	3900	0.99999
D	4	1000	10	5200	0.99999
E	5	1000	10	6500	0.99999

Table 4.4.: Input scenarios

Since the values for $\mathbb{\Lambda}_{\bar{n}_c^*}(D)$ are numbers which are very close to 1, the difference $1 - \mathbb{\Lambda}_{\bar{n}_c^*}(D)$ results in numbers that are more convenient to present by using scientific notation. Therefore, instead of presenting the values for $\mathbb{\Lambda}_{\bar{n}_c^*}(D)$, we present the values for the expression $\bar{\mathbb{\Lambda}}_{\bar{n}_c^*}(D) = 1 - \mathbb{\Lambda}_{\bar{n}_c^*}(D)$. One observes that lower values for $\bar{\mathbb{\Lambda}}_{\bar{n}_c^*}(D)$ represent a better solution, *i.e.* $\mathbb{\Lambda}_{\bar{n}_c^*}(D)$ is higher.

For P1, *i.e.* comparing $\mathbb{\Lambda}_{\bar{n}_c^*}(D)$ obtained from the four approaches, the results are summarized in Table 4.5. For the Local Optimization approach the results are obtained as follows. For each scenario, first we calculate the local deadline D' for each job by dividing the global deadline D with the number of jobs in the set m, *i.e.* $D' = \frac{D}{m}$. Next, we obtain the optimal number of checkpoints n_c^* that provides the maximal LoC with respect to the local deadline D', *i.e* $\Lambda_{n_c^*}(D')$, as shown in Section 4.1.4. Once n_c^* is calculated, we apply the expression in Eq. (4.54) to calculate $\mathbb{\Lambda}_{\bar{n}_c^*}(D)$. Important to note is that all

jobs in the set use the same number of checkpoints, and therefore the reported checkpoint assignment $\bar{n}_c^*$ consists of m elements, all with the same value n_c^*.

For the Single Large Job approach, the results are obtained by computing the optimal number of checkpoints n_c^*, for a single job with a processing time $T' = m \times T$, that provides the maximal LoC with respect to the global deadline D, *i.e.* $\Lambda_{n_c^*}(D)$, as shown in Section 4.1.4. Once n_c^* is calculated, the reported checkpoint assignment $\bar{n}_c^*$ is evaluated as a vector where all elements have the same value $\lfloor \frac{n_c^*}{m} \rfloor$. In Table 4.5 we present the highest LoC that can be achieved by the Single Large Job approach. However, we cannot always rely on the results obtained from this approach. Observe the reported results for Scenario B and Scenario E (marked with asterisk). For example, when running the experiment for Scenario E, the Single Large Job approach reports the highest LoC which is obtained when using $n_c^* = 92$ checkpoints for a single large job which is an equivalent of five jobs running in a sequence. Using $n_c^* = 92$ checkpoints implies existence of execution segments that consist of portions of two different jobs, due to the fact that 92 is not a multiple of five (the number of jobs in the set).

Scenario	Approach	$\bar{\mathbb{\Lambda}}_{\bar{n}_c^*}(D)$ and $\bar{n}_c^*$
A	LO	$n_c^* = 25 \rightarrow \bar{n}_c^* = [25, 25]$
		4.863650178E-35
	SLJ	$n_c^{**} = 50 \rightarrow \bar{n}_c^* = [25, 25]$
		4.863650178E-35
	EXS	$\bar{n}_c^* = [25, 25]$
		4.863650178E-35
	SES	$\bar{n}_c^* = [25, 25]$
		4.863650178E-35
B	LO	$n_c^* = 25 \rightarrow \bar{n}_c^* = [25, 25]$
		1.131502348E-14
	SLJ	$n_c^{**} = 43 \rightarrow \bar{n}_c^* = [21, 21]$
		1.221914117E-19*
	EXS	$\bar{n}_c^* = [14, 19]$
		8.767754710E-20
	SES	$\bar{n}_c^* = [14, 19]$
		8.767754710E-20

to be continued on next page

Scenario	Approach	$\bar{\lambda}_{\bar{n}_c^*}(D)$ and $\bar{n}_c^*$
C	LO	$n_c^* = 25 \rightarrow \bar{n}_c^* = [25, 25, 25]$
		5.842467599E-19
	SLJ	$n_c^{**} = 60 \rightarrow \bar{n}_c^* = [20, 20, 20]$
		8.259693303E-29
	EXS	$\bar{n}_c^* = [13, 16, 16]$
		5.768673354E-29
	SES	$\bar{n}_c^* = [13, 16, 16]$
		5.768673354E-29
D	LO	$n_c^* = 25 \rightarrow \bar{n}_c^* = [25, 25, 25, 25]$
		3.013298280E-23
	SLJ	$n_c^{**} = 76 \rightarrow \bar{n}_c^* = [19, 19, 19, 19]$
		5.945230027E-38
	EXS	$\bar{n}_c^* = [13, 14, 14, 18]$
		5.231888327E-38
	SES	$\bar{n}_c^* = [13, 14, 14, 18]$
		5.231888327E-38
E	LO	$n_c^* = 25 \rightarrow \bar{n}_c^* = [25, 25, 25, 25, 25]$
		1.563182901E-27
	SLJ	$n_c^{**} = 92 \rightarrow \bar{n}_c^* = [18, 18, 18, 18, 18]$
		4.422693488E-47*
	EXS	$\bar{n}_c^* = [14, 14, 14, 17, 17]$
		4.0544649088E-47
	SES	$\bar{n}_c^* = [14, 14, 14, 17, 17]$
		4.0544649088E-47

Table 4.5.: Comparison of $\bar{\lambda}_{\bar{n}_c^*}(D)$ for the Local Optimization (LO), the Single Large Job (SLJ), the Exhaustive Search (EXS), and the Semi-Exhaustive Search (SES) approach

We cannot have such execution segments, which leads to the fact that we cannot use $n_c^* = 92$. Instead, we use $n_c = 90$ for the single large job, which then implies that each of the five jobs uses $n_c = 18$ checkpoints. Note that using $n_c = 90$ will result in a lower $\lambda_{\bar{n}_c}(D)$ than the one reported (the highest

LoC is achieved for $n_c^* = 92$). The same discussion applies for Scenario B, where the highest LoC obtained from the Single Large Job approach is computed for $n_c^* = 43$. However, just for comparison purposes in Table 4.5 we report the highest LoC achieved by the Single Large Job, and mark the infeasible solutions with an asterisk (Scenario B and Scenario E). One observes that, in general, the Single Large Job approach does not achieve the maximal LoC (compared with Exhaustive Search and Semi-Exhaustive Search approach), even for the infeasible solutions.

For the Exhaustive Search approach the results are obtained by computing $\mathbb{\Lambda}_{\bar{n}_c}(D)$, using Eq. (4.60), for all *valid* checkpoint assignments $\bar{n}_c$. Once all *valid* checkpoint assignments are exhausted, the optimal checkpoint assignment $\bar{n}_c^*$ is reported along with $\mathbb{\Lambda}_{\bar{n}_c^*}(D)$ in Table 4.5.

Finally, for the Semi-Exhaustive Search approach, similar to the Exhaustive Search approach, the results are obtained by computing $\mathbb{\Lambda}_{\bar{n}_c}(D)$, using Eq. (4.60), for a subset of *valid* checkpoint assignments $\bar{n}_c$ as discussed in Section 4.2.4. The optimal checkpoint assignment $\bar{n}_c^*$ along with $\mathbb{\Lambda}_{\bar{n}_c^*}(D)$ are reported in Table 4.5.

With the result set presented in Table 4.5 we want to point out that both the Exhaustive Search and the Semi-Exhaustive Search approach provide the same results for all input scenarios, and further both of these approaches obtain the maximal $\mathbb{\Lambda}_{\bar{n}_c^*}(D)$. From Table 4.5, we observe that sometimes even the Local Optimization and the Single Large Job approach may provide the maximal $\mathbb{\Lambda}_{\bar{n}_c^*}(D)$ (observe the results for Scenario A). However, we cannot guarantee that the results obtained from these two approaches are optimal for a random input scenario (observe the results for Scenario B, C, D and E). Next, we present results for the second experiment P2.

For P2, we compare the computation time for the Exhaustive Search approach T_{EXS} against the computation time for the Semi-Exhaustive Search approach T_{SES}. We have already shown in the previous result set that both approaches provide the same results for all input scenarios. Important to note is that for this experiment we adjust the Exhaustive Search approach such that it does not explore equivalent checkpoint assignments. The computation time for both approaches and for each of the input scenarios is presented in Table 4.6. In addition to the reported computation time for both approaches, in Table 4.6 we report the reduction ratio. The reduction ratio shows the improvement of the Semi-Exhaustive Search over the Exhaustive Search approach in terms of computation time, and it is calculated as the ratio T_{EXS}/T_{SES}. As observed from Table 4.6, T_{EXS} grows rapidly as the number of jobs increases. We also observe from Table 4.6 that T_{SES} also increases with the number of jobs. However, T_{SES} is always lower than T_{EXS}. This is clearly visible when we examine input scenarios that consist of larger sets of jobs. One particular result that we want to point out is the result obtained for Sce-

Scenario	T_{EXS}	T_{SES}	Reduction ratio $\left(\frac{T_{EXS}}{T_{SES}}\right)$
A	1965 *ms*	1162 *ms*	1.691
B	1396 *ms*	1023 *ms*	1.365
C	9131 *ms*	1906 *ms*	4.791
D	623710 *ms*	37918 *ms*	16.449
E	41380904 *ms*	2321786 *ms*	17.823

Table 4.6.: Comparison of the computation time for the Exhaustive Search (EXS) and the Semi-Exhaustive Search (SES) approach

nario E in Table 4.6, where we observe that T_{SES} is almost 18 times lower than T_{EXS} (the reduction ratio is 17.823). The reason that makes both of these approaches very time-consuming is that both approaches need to execute many complex floating-point operations with a very high precision. However, as the Semi-Exhaustive Search explores only a subset of all *valid* checkpoint assignments, the computation time for this approach is drastically reduced when compared to the Exhaustive Search approach.

5

Guaranteed Completion Time

So far, we discussed that minimizing the AET is more important for soft RTS while maximizing the LoC is more important for hard RTS. However, a designer might be given a system specification which includes some reliability requirements. In such scenario, instead of optimizing RRC with the goal to minimize the AET or maximize the LoC, it becomes important to optimize RRC such that the minimal completion time is obtained while given reliability requirements are satisfied. With respect to this we introduce the term Guaranteed Completion Time (GCT_δ) that refers to a completion time which satisfies a given LoC requirement δ. Introducing the GCT_δ opens another interesting problem to consider, *i.e.* optimize RRC with the goal to minimize the GCT_δ and we discuss that problem in this chapter. The chapter is organized as follows. The problem formulation is stated in Section 5.1. Definition and properties of GCT_δ are provided in Section 5.2. A method for minimizing GCT_δ is presented in Section 5.3. Finally, some experimental results are presented in Section 5.4.

5.1. PROBLEM FORMULATION

In this section we discuss the following problem. Given the following inputs:

- a job with a processing time T,
- an LoC requirement δ,
- a checkpointing overhead τ, and
- a probability P_T that no soft errors occur in a processing node within an interval of length T,

find the optimal number of checkpoints such that the Guaranteed Completion Time (GCT_δ), i.e. the completion time that satisfies the given LoC requirement δ, is minimized.

5.2. DEFINITION AND PROPERTIES

We introduce the term *Guaranteed Completion Time* (GCT_δ) which is a completion time ${}^{n_c}t_k$ that is guaranteed with a given LoC requirement δ. Hence, a completion time ${}^{n_c}t_k$ represents a GCT_δ, if the following condition holds:

$$\lambda_{n_c}(k) \geq \delta \tag{5.1}$$

Observe that in Eq. (5.1) we are using the notation $\lambda_{n_c}(k)$ since we evaluate the LoC with respect to a given instance of the completion time ${}^{n_c}t_k$. The condition in Eq. (5.1) can be satisfied for many different pairs of values (k,n_c). The reason for this is a consequence of the properties of the expression which is used to calculate the LoC, in particular **Theorem 2** and **Theorem 3** (see Section 4.1.3 in Chapter 4). According to **Theorem 2**, if for a given $n_c^\dagger$ and $k^\dagger$ the condition in Eq. (5.1) is satisfied, then the same condition will be satisfied for any combination of $n_c^\dagger$ and k where $k > k^\dagger$, *i.e.* Eq. (5.1) is satisfied for all pairs $(k, n_c^\dagger)$ where $k > k^\dagger$. Hence, for any pair $(n_c^\dagger, k)$ where $k > k^\dagger$, the condition $\lambda_{n_c^\dagger}(k) > \lambda_{n_c^\dagger}(k^\dagger)$ is satisfied (this is consequence of **Theorem 2**), which means that also Eq. (5.1) is satisfied. On the other hand, according to **Theorem 3**, if for a given $n_c^\dagger$ and $k^\dagger$ the condition in Eq. (5.1) is satisfied, then the same condition will be satisfied for all pairs $(k^\dagger, n_c)$ where $n_c > n_c^\dagger$, *i.e.* $\lambda_{n_c}(k^\dagger) > \lambda_{n_c^\dagger}(k^\dagger)$ when $n_c > n_c^\dagger$. Thus, as a result of **Theorem 2** and **Theorem 3**, we conclude that the condition in Eq. (5.1) can be satisfied for many different pairs of values (k,n_c).

While Eq. (5.1) can be satisfied for many different pairs of values (k,n_c), there exists a lower bound k_δ such that Eq. (5.1) can be satisfied only if $k \geq k_\delta$. This comes as a consequence of **Theorem 4** (see Section 4.1.3 in Chapter 4). According to **Theorem 4**, for a given $k^\dagger$, $\lambda_{n_c}(k^\dagger)$ cannot be higher than $\bar{\lambda}_{k^\dagger}$ for any n_c. Thus, if $\bar{\lambda}_{k^\dagger} < \delta$, the condition in Eq. (5.1) cannot be satisfied for any pair of values $(k^\dagger, n_c)$. This implies, that the condition in Eq. (5.1) can be satisfied only for such values of k where $\bar{\lambda}_k \geq \delta$. We denote with k_δ, the lower bound (lowest integer value) for k that satisfies the condition $\bar{\lambda}_k \geq \delta$. Obtaining k_δ is important as Eq. (5.1) can be satisfied only if $k \geq k_\delta$.

Observe that the condition $k \geq k_\delta$ is just necessary and not sufficient condition to satisfy Eq. (5.1). The condition $k \geq k_\delta$ states that it is required k to be greater than or equal to k_δ in order to satisfy Eq. (5.1). However, not all combinations of n_c and $k \geq k_\delta$ satisfy Eq. (5.1). Instead, for any $k \geq k_\delta$ there exists a value for n_c, denoted with ${}^g_k n_c$, such that for any $n_c \geq {}^g_k n_c$ and

$k \geq k_\delta$ Eq. (5.1) is satisfied. As there is no closed-form expression to obtain $\lambda_{n_c}(k)$ (see Eq. (4.11)), it is not possible for a given k to directly compute ${}^{g}_{k}n_c$. However, it is possible to conclude that a relation exists between k and ${}^{g}_{k}n_c$. The relation between k and ${}^{g}_{k}n_c$ is such that ${}^{g}_{k}n_c$ decreases as k increases.

To prove that such relation exists, we make use of **Theorem 2** and **Theorem 3**. Let us assume that for a given $k^\dagger > k_\delta$ we obtain $n_c^\dagger = {}^{g}_{k^\dagger}n_c$, such that the following relation holds:

$$\lambda_{n_c^\dagger}(k^\dagger) = \delta \tag{5.2}$$

Hence, the pair $(k^\dagger, n_c^\dagger)$ satisfies Eq. (5.1) with the "equality" sign as shown in Eq. (5.2). According to **Theorem 2**, for a given $k^\ddagger$, such that $k^\ddagger > k^\dagger$, the following relation holds:

$$\lambda_{n_c^\dagger}(k^\ddagger) > \lambda_{n_c^\dagger}(k^\dagger) \tag{5.3}$$

As a result of **Theorem 2**, the pair $(k^\ddagger, n_c^\dagger)$ will also satisfy Eq. (5.1). However, the pair $(k^\ddagger, n_c^\dagger)$ satisfies Eq. (5.1) with the strictly "greater than" sign, *i.e.*:

$$\lambda_{n_c^\dagger}(k^\ddagger) > \delta \tag{5.4}$$

According to **Theorem 3**, for any $n_c < n_c^\dagger$ the following condition holds:

$$\lambda_{n_c}(k^\ddagger) < \lambda_{n_c^\dagger}(k^\ddagger) \tag{5.5}$$

However, as the right-hand side of Eq. (5.5) is strictly greater than δ (see Eq. (5.4)), it is still possible to obtain an $n_c^\ddagger$ that is lower than $n_c^\dagger = {}^{g}_{k^\dagger}n_c$, such that the pair $(k^\ddagger, n_c^\ddagger)$ satisfies Eq. (5.1). Assuming that all the pairs $(k^\ddagger, n_c)$ where $n_c < n_c^\ddagger$ do not satisfy Eq. (5.1), leads to ${}^{g}_{k^\ddagger}n_c = n_c^\ddagger$. Since $n_c^\ddagger$ is lower than $n_c^\dagger$, we conclude that ${}^{g}_{k^\ddagger}n_c < {}^{g}_{k^\dagger}n_c$ while assuming that $k^\ddagger > k^\dagger$. By this, we prove that ${}^{g}_{k}n_c$ decreases as k increases.

So far, we have shown that the necessary condition, given in Eq. (5.1), for a completion time ${}^{n_c}t_k$ to represent a GCT_δ can be satisfied for many different pairs of values (k, n_c). However, using different values for n_c and k affects the completion time ${}^{n_c}t_k$. Next, we study how n_c and k affect the completion time.

As shown in the previous chapter, the completion time ${}^{n_c}t_k$, given in Eq. (4.3), depends on the number of checkpoints n_c and the number of erroneous execution segments (number of re-executions) k. For convenience, we restate the expression for the completion time ${}^{n_c}t_k$, *i.e.*:

$${}^{n_c}t_k = T + n_c \times \tau + k \times (\frac{T}{n_c} + \tau) \tag{5.6}$$

For a fixed number of checkpoints $n_c^\dagger$, the completion time monotonically increases as k increases, *i.e.* ${}^{n_c^\dagger}t_{k+1} > {}^{n_c^\dagger}t_k$. On the other hand, for a fixed number of re-executions $k^\dagger > 0$, there exists an optimal number of checkpoints ${}_{k^\dagger}n_c^*$, such that for any given n_{c_1} and n_{c_2} the following two relations hold: if $n_{c_1} < n_{c_2} < {}_{k^\dagger}n_c^*$, then ${}^{n_{c_1}}t_k > {}^{n_{c_2}}t_k$, and if $n_{c_1} > n_{c_2} > {}_{k^\dagger}n_c^*$, then ${}^{n_{c_1}}t_k < {}^{n_{c_2}}t_k$. This means that for a fixed number of re-executions $k^\dagger$, where $k^\dagger > 0$, the completion time decreases as the number of checkpoints n_c increases up to a certain point, *i.e.* as n_c increases up to ${}_{k^\dagger}n_c^*$. However, increasing the number of checkpoints n_c beyond ${}_{k^\dagger}n_c^*$ increases the completion time as well. In Figure 5.1, we illustrate the completion time as a function of n_c and each curve in Figure 5.1 represents a completion time corresponding to a fixed k. As can be seen from Figure 5.1, for each $k > 0$ there exists an optimal number of checkpoints ${}_kn_c^*$ such that the minimal completion time for the given k is achieved when ${}_kn_c^*$ checkpoints are used.

To find, for a given $k > 0$, the optimal number of checkpoints ${}_kn_c^*$ that results in the minimal completion time, we take the first derivative of the expression that represents the completion time, *i.e.* Eq. (5.6), with respect to n_c and set it to be equal to zero. Taking the first derivative of ${}^{n_c}t_k$ with respect to n_c and setting it to be equal to zero results in the following:

$$
\begin{aligned}
&\frac{\mathrm{d}^{n_c}t_k}{\mathrm{d}n_c} = 0 \\
&\Rightarrow \frac{\mathrm{d}}{\mathrm{d}n_c}\left(T + n_c \times \tau + k \times (\frac{T}{n_c} + \tau)\right) = 0 \\
&\Rightarrow \frac{\mathrm{d}}{\mathrm{d}n_c}(T) + \frac{\mathrm{d}}{\mathrm{d}n_c}(n_c \times \tau) + \frac{\mathrm{d}}{\mathrm{d}n_c}\left(k \times \frac{T}{n_c}\right) + \frac{\mathrm{d}}{\mathrm{d}n_c}(k \times \tau) = 0 \\
&\Rightarrow \tau - k \times \frac{T}{{n_c}^2} = 0 \\
&\Rightarrow {}_kn_c^* = \sqrt{k \times \frac{T}{\tau}}
\end{aligned}
\tag{5.7}
$$

Observe that the right-hand side of the expression in Eq. (5.7) is evaluated as a real number. However, since ${}_kn_c^*$ represents a number of checkpoints, only integer values are allowed. Rounding ${}_kn_c^*$, given in Eq. (5.7), to the closest integer value does not always provide the optimal number of checkpoints that leads to the minimal completion time, for a given $k > 0$ (a discussion on this is presented in Appendix D). To verify that we obtain the optimal ${}_kn_c^*$, we need to compare the completion times for the upper and the lower integer bound of the expression given in Eq. (5.7). Thus, we obtain the following expression to compute, for a given $k > 0$, the optimal number of checkpoints ${}_kn_c^*$ that

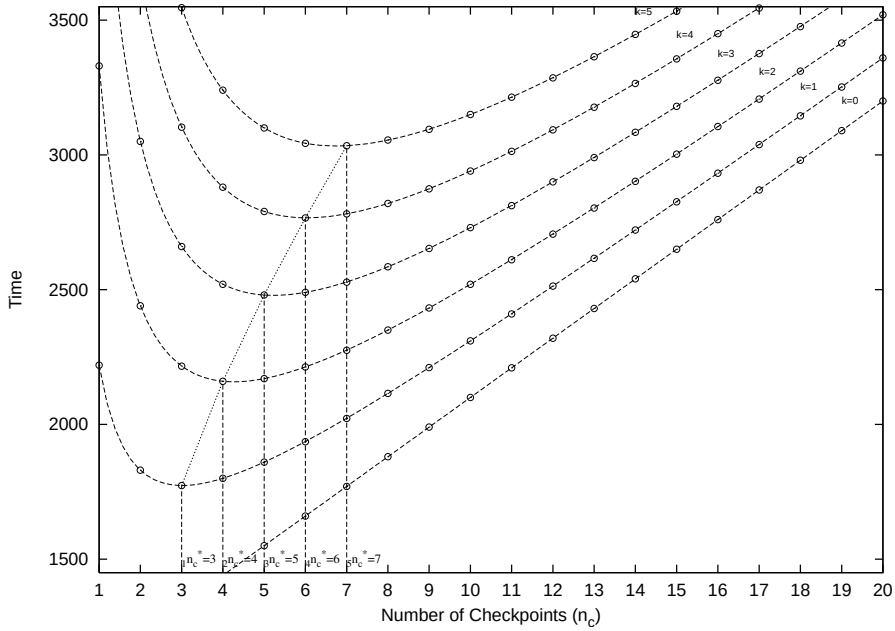


Figure 5.1.: Illustration of the completion time ${}^{n_c}t_k$ as a function of n_c, for given $T = 1000$ t.u. (time units), $\tau = 110$ t.u., and $k \in [0, 5]$

results in the minimal completion time:

$$
\begin{aligned}
n_{c_L} &= \left\lfloor \sqrt{k \times \frac{T}{\tau}} \right\rfloor \\
n_{c_U} &= \left\lceil \sqrt{k \times \frac{T}{\tau}} \right\rceil \\
{}_k n_c^* &= \begin{cases} n_{c_L}, \text{ if } {}^{n_{c_L}}t_k < {}^{n_{c_U}}t_k \\ n_{c_U}, \text{ if } {}^{n_{c_U}}t_k \leq {}^{n_{c_L}}t_k \end{cases}
\end{aligned} \tag{5.8}
$$

When the number of re-executions k is greater than zero, there exists an optimal number of checkpoints ${}_k n_c^*$ that, for the given k, results in the minimal completion time and ${}_k n_c^*$ is computed as given in Eq. (5.8). However, the number of re-executions k is also allowed to be zero. When $k = 0$, the completion time monotonically increases along as n_c increases (observe the linear trend for the curve $k = 0$ in Figure 5.1). For $k = 0$, the minimal (the lowest) completion time is achieved for the lowest value that can be assigned to n_c,

i.e. $n_c = 1$.

Observe that for $k = 0$, Eq. (5.8) evaluates the optimal number of checkpoints ${}_0n_c^*$ as zero, *i.e.* ${}_0n_c^* = 0$. However, we showed in the previous paragraph, that when $k = 0$ the minimal completion time is achieved when a single checkpoint is used, *i.e.* ${}_0n_c^* = 1$. Therefore, we need to adjust Eq. (5.8) and use the following expression to calculate the optimal number of checkpoints ${}_kn_c^*$, for any given $k \geq 0$:

$$
\begin{aligned}
n_{c_L} &= \left\lfloor \sqrt{k \times \frac{T}{\tau}} \right\rfloor \\
n_{c_U} &= \left\lceil \sqrt{k \times \frac{T}{\tau}} \right\rceil \\
{}_kn_c^* &= \begin{cases} 1, & k = 0 \\ n_{c_L}, & \text{if } {}^{n_{c_L}}t_k < {}^{n_{c_U}}t_k, k > 0 \\ n_{c_U}, & \text{if } {}^{n_{c_U}}t_k \leq {}^{n_{c_L}}t_k, k > 0 \end{cases}
\end{aligned}
\tag{5.9}
$$

So far, we showed that for a fixed n_c, the completion time increases as k increases and we showed that for a fixed k there exists an optimal ${}_kn_c^*$ that provides the minimal completion time for the given k. Another important observation is that for each pair $(k, {}_kn_c^*)$ the completion time ${}^{k n_c^*}t_k$ increases as k increases. This is illustrated in Figure 5.1 (observe the line that connects the optimal points of the curves illustrated in Figure 5.1). By observing the optimal points for each of the curves illustrated in Figure 5.1, we see an increasing trend. The increasing trend is also shown with the following equation:

$$
\begin{aligned}
{}^{k n_c^*}t_k &= T + \sqrt{k\frac{T}{\tau}} \times \tau + k \times \left(\frac{T}{\sqrt{k \times \frac{T}{\tau}}} + \tau \right) \\
&= T + 2 \times \sqrt{k \times T \times \tau} + k \times \tau
\end{aligned}
\tag{5.10}
$$

The expression in Eq. (5.10) represents the completion time for the pair of values $(k, {}_kn_c^*)$. Since we assume that the processing time T and the checkpointing overhead τ are given, Eq. (5.10) represents a function that depends only on the number of re-executions k. To show that this function, *i.e.* Eq. (5.10), is monotonically increasing with k it is necessary to show that the first derivative of Eq. (5.10) is always positive. By calculating the first derivative of Eq. (5.10) with respect to k we get:

$$
\sqrt{\frac{T \times \tau}{k}} + \tau \tag{5.11}
$$

The expression in Eq. (5.11) is always evaluated as a positive number, which means that the expression in Eq. (5.10) is a function that monotonically increases with k. Since Eq. (5.10) represents the minimal completion time with respect to k, *i.e.* ${}^{k}{}^{n_c^*}t_k$, we conclude that by increasing k, the minimal completion time ${}^{k}{}^{n_c^*}t_k$ increases as well. The increasing trend is also shown in Figure 5.1 (observe the line which connects the optimal points of the curves for different values of k).

As shown in Figure 5.1, for any given $k^\dagger > 0$, the completion time ${}^{n_c}t_{k^\dagger}$ reaches a minimum when the number of checkpoints n_c is equal to ${}_{k}n_c^*$ and for any other n_c value, ${}^{n_c}t_{k^\dagger}$ increases. However, even though ${}^{n_c}t_{k^\dagger}$ increases, there exists a range of values for n_c, such that the completion time ${}^{n_c}t_{k^\dagger}$ is still lower than the minimal completion time ${}^{k}{}^{n_c^*}t_k$ that is achieved for any $k > k^\dagger$. This shows that in order to obtain a lower completion time, it is also important to consider, for a given k, other values for n_c besides the optimal ${}_{k}n_c^*$.

Next, we show how to obtain the range of values for n_c, denoted with $[{}^{k^\ddagger}_{k^\dagger}n_{c_L}, {}^{k^\ddagger}_{k^\dagger}n_{c_U}]$, such that for given $k^\dagger < k^\ddagger$ the following relation holds:

$$ {}^{n_c}t_{k^\dagger} \leq {}^{k^\ddagger}{}^{n_c^*}t_{k^\ddagger} \tag{5.12} $$

By using Eq. (5.6), we re-write Eq. (5.12) as:

$$
\begin{aligned}
& T + n_c \times \tau + k^\dagger \times \left(\frac{T}{n_c} + \tau\right) \leq T + {}_{k^\ddagger}n_c^* \times \tau + k^\ddagger \times \left(\frac{T}{{}_{k^\ddagger}n_c^*} + \tau\right) \\
\Leftrightarrow\ & n_c \times \tau + k^\dagger \times \left(\frac{T}{n_c} + \tau\right) \leq {}_{k^\ddagger}n_c^* \times \tau + k^\ddagger \times \left(\frac{T}{{}_{k^\ddagger}n_c^*} + \tau\right) \\
\Leftrightarrow\ & n_c \times \tau + \left(k^\dagger - {}_{k^\ddagger}n_c^* - k^\ddagger\right) \times \tau + k^\dagger \times \frac{T}{n_c} - k^\ddagger \times \frac{T}{{}_{k^\ddagger}n_c^*} \leq 0 \\
\Leftrightarrow\ & \tau \times {n_c}^2 + \underbrace{\left(\left(k^\dagger - {}_{k^\ddagger}n_c^* - k^\ddagger\right) \times \tau - k^\ddagger \times \frac{T}{{}_{k^\ddagger}n_c^*}\right)}_{C} \times n_c + k^\dagger \times T \leq 0 \\
\Leftrightarrow\ & \tau \times {n_c}^2 + C \times n_c + k^\dagger \times T \leq 0
\end{aligned} \tag{5.13}
$$

The left-hand side of Eq. (5.13) represents a quadratic function of n_c. Furthermore, this function reaches a minimum point because the second derivative, which is evaluated as τ, is positive, *i.e.* $\tau > 0$. The quadratic function given in Eq. (5.13) is evaluated as either negative or equal to zero for any value of n_c that belongs to the range $[n_{c_1}, n_{c_2}]$ where n_{c_1} and n_{c_2} represent the roots of the function. Observe that the function can be negative or equal to zero if and only if the roots of the function are evaluated as real numbers, *i.e.* $n_{c_1}, n_{c_2} \in \mathbb{R}$. Otherwise, if $n_{c_1}, n_{c_2} \notin \mathbb{R}$ (the roots are complex numbers), the function is evaluated as positive for any value of n_c. The roots of the quadratic

function given in Eq. (5.13) are real numbers, because the discriminant, which is given in Eq. (5.14), is positive.

$$C^2 - 4 \times k^{\dagger} \times T \times \tau = \\ = \left(\left(k^{\dagger} - {}_{k^{\ddagger}}n_c^* - k^{\ddagger} \right) \times \tau - k^{\ddagger} \times \frac{T}{{}_{k^{\ddagger}}n_c^*} \right)^2 - 4 \times k^{\dagger} \times T \times \tau \tag{5.14}$$

Next, we prove that the discriminant, *i.e.* Eq. (5.14), is evaluated as positive. Replacing ${}_{k^{\ddagger}}n_c^*$, in Eq. (5.14), with Eq. (5.7) results in the following expression:

$$\left(\left(k^{\dagger} - \sqrt{k^{\ddagger} \times \frac{T}{\tau}} - k^{\ddagger} \right) \times \tau - k^{\ddagger} \times \frac{T}{\sqrt{k^{\ddagger} \times \frac{T}{\tau}}} \right)^2 - 4 \times k^{\dagger} \times T \times \tau = \\ = \left(\left(k^{\dagger} - k^{\ddagger} \right) \times \tau - 2 \times \sqrt{k^{\ddagger} \times T \times \tau} \right)^2 - 4 \times k^{\dagger} \times T \times \tau \\ = \left(\left(k^{\dagger} - k^{\ddagger} \right) \times \tau \right)^2 + 4 \times \left(k^{\ddagger} - k^{\dagger} \right) \times \sqrt{k^{\ddagger} \times T \times \tau} + 4 \times \left(k^{\ddagger} - k^{\dagger} \right) \times T \times \tau \tag{5.15}$$

The expression in Eq. (5.15) consists of three terms and all these terms are greater than zero. The first term $\left(\left(k^{\dagger} - k^{\ddagger} \right) \times \tau \right)^2$ is always positive because it is obtained as a power of two. The second term $4 \times \left(k^{\ddagger} - k^{\dagger} \right) \times \sqrt{k^{\ddagger} \times T \times \tau}$ and the third term $4 \times \left(k^{\ddagger} - k^{\dagger} \right) \times T \times \tau$ are evaluated as positive due to that $k^{\ddagger}$ is greater than $k^{\dagger}$. Since all the terms in Eq. (5.15) are greater than zero, the discriminant of the quadratic function is also greater than zero. This implies that the roots n_{c_1} and n_{c_2} will be evaluated as real numbers and furthermore, Eq. (5.13) will be satisfied for any n_c value that belongs to the range $[n_{c_1}, n_{c_2}]$. The roots of the quadratic function represented with the left-hand side expression of Eq. (5.13) are evaluated with the following equations:

$$n_{c_1} = \frac{-C - \sqrt{C^2 - 4 \times k^{\dagger} \times T \times \tau}}{2 \times \tau} \tag{5.16}$$

$$n_{c_2} = \frac{-C + \sqrt{C^2 - 4 \times k^{\dagger} \times T \times \tau}}{2 \times \tau} \tag{5.17}$$

The constant C in Eq. (5.16) and Eq. (5.17) is evaluated with the following expression:

$$C = \left(k^{\dagger} - {}_{k^{\ddagger}}n_c^* - k^{\ddagger} \right) \times \tau - k^{\ddagger} \times \frac{T}{{}_{k^{\ddagger}}n_c^*} \tag{5.18}$$

As $k^{\dagger} < k^{\ddagger}$, the constant C (see Eq. (5.18)) will be always evaluated as a negative value. Since C is negative, both roots n_{c_1} and n_{c_2} will be evaluated as positive (see Eq. (5.16) and Eq. (5.17)).

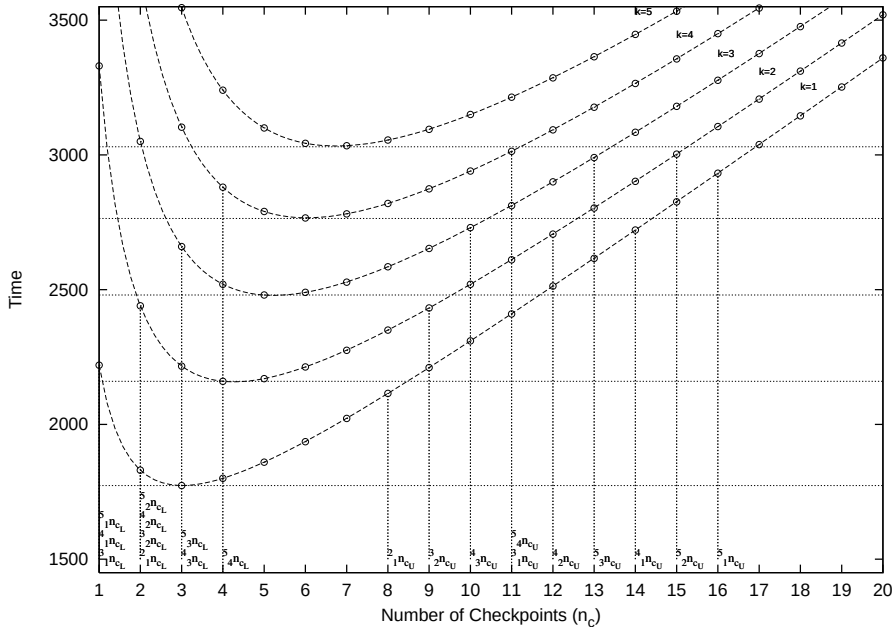


Figure 5.2.: Illustration of ${}^{k^\ddagger}_{k^\dagger}n_{c_L}$ and ${}^{k^\ddagger}_{k^\dagger}n_{c_U}$, for given $T = 1000$ t.u., $\tau = 110$ t.u., and $k \in [1,5]$

Finally, Eq. (5.13) will be satisfied for any $n_c \in [n_{c_1}, n_{c_2}]$. Since only integer values can be assigned to n_c, we define the boundaries of this range with the following expressions:

$$ {}^{k^\ddagger}_{k^\dagger}n_{c_L} = \left\lceil \frac{-C - \sqrt{C^2 - 4 \times k^\dagger \times T \times \tau}}{2 \times \tau} \right\rceil \tag{5.19} $$

$$ {}^{k^\ddagger}_{k^\dagger}n_{c_U} = \left\lfloor \frac{-C + \sqrt{C^2 - 4 \times k^\dagger \times T \times \tau}}{2 \times \tau} \right\rfloor \tag{5.20} $$

To conclude, for any given $k^\dagger < k^\ddagger$, there exists a range of values for n_c, *i.e.* $[{}^{k^\ddagger}_{k^\dagger}n_{c_L}, {}^{k^\ddagger}_{k^\dagger}n_{c_U}]$, such that Eq. (5.12) is satisfied. The boundaries ${}^{k^\ddagger}_{k^\dagger}n_{c_L}$ and ${}^{k^\ddagger}_{k^\dagger}n_{c_U}$ are evaluated with Eq. (5.19) and Eq. (5.20) respectively.

We illustrate the notations ${}^{k^\ddagger}_{k^\dagger}n_{c_L}$ and ${}^{k^\ddagger}_{k^\dagger}n_{c_U}$ in Figure 5.2, where we show the completion time for $k \in [1,5]$ as a function of n_c. As shown in Figure 5.2, for any n_c that belongs to the range $[{}^5_2n_{c_L} = 2, {}^5_2n_{c_U} = 15]$, the completion time along the curve $k = 2$ is lower than the minimum point of the curve $k = 5$.

We summarize this section with the following conclusions:

- Many instances of the completion time ${}^{n_c}t_k$ may satisfy a given LoC requirement δ and thus, represent a GCT_δ;
- A GCT_δ is always associated with a pair of values (k, n_c);
- For a given δ, only instances of the completion time where $k > k_\delta$ may represent a GCT_δ;
- For any given $k \geq k_\delta$ there exists a ${}^{g}_{k}n_c$, such that for any $n_c \geq {}^{g}_{k}n_c$ all instances (k, n_c) represent a GCT_δ; and
- ${}^{g}_{k}n_c$ decreases as k increases

In Figure 5.3 we illustrate GCT_δ for the following inputs: $T = 1000$ t.u., $\tau = 110$ t.u., $P_T = 0.9$, and $\delta = 0.99999998$. The filled circles shown in Figure 5.3 represent instances of the completion time which represent a GCT_δ, while the empty circles are instances of the completion time which do not represent a GCT_δ. As shown in Figure 5.3 many instances of the completion time may represent a GCT_δ. However, only instances for $k \geq k_\delta$ may represent a GCT_δ (observe $k = 6$ in Figure 5.3). Further, in Figure 5.3 we illustrate the notations ${}^{g}_{k}n_c$. As shown in Figure 5.3, ${}^{g}_{k}n_c$ decreases as k increases. For example, ${}^{g}_{k}n_c$ for $k = 6$ is 9 (observe ${}^{g}_{6}n_c = 9$ in Figure 5.3) and it is larger than ${}^{g}_{k}n_c$ for $k = 8$ which is 2 (observe ${}^{g}_{8}n_c = 2$ in Figure 5.3).

For a given $k \geq k_\delta$, the lowest GCT_δ with respect to k is achieved for a value of n_c which is either ${}_{k}n^*_c$ or ${}^{g}_{k}n_c$. If for a given k, the following condition holds ${}^{g}_{k}n_c \leq {}_{k}n^*_c$, then the lowest GCT_δ with respect to k is obtained for the pair of values $(k, {}_{k}n^*_c)$. Otherwise, if for a given k, the condition ${}^{g}_{k}n_c \leq_k n^*_c$ is not satisfied, then the lowest GCT_δ for the given k is obtained for the pair of values $(k, {}^{g}_{k}n_c)$.

If for all $k \geq k_\delta$, the condition ${}^{g}_{k}n_c \leq {}_{k}n^*_c$ is satisfied, then the minimal GCT_δ is obtained for the pair of values $(k_\delta, {}_{k_\delta}n^*_c)$. However, there is no straightforward relation to obtain the minimal GCT_δ if the condition ${}^{g}_{k}n_c \leq {}_{k}n^*_c$ does not hold for all $k \geq k_\delta$. For that reason, in the next section, we propose a method to obtain the minimal GCT_δ and the optimal number of checkpoints $n^*_{c_\delta}$.

5.3. MINIMIZING GUARANTEED COMPLETION TIME

In this section, we present an optimization method for RRC where the optimization goal is to find the optimal number of checkpoints $n^*_{c_\delta}$ such that for a given LoC requirement δ, the minimal GCT_δ is achieved. Given the following parameters: a processing time T, a checkpointing overhead τ, a probability P_T that no errors occur in a processing node within an interval of length T, and

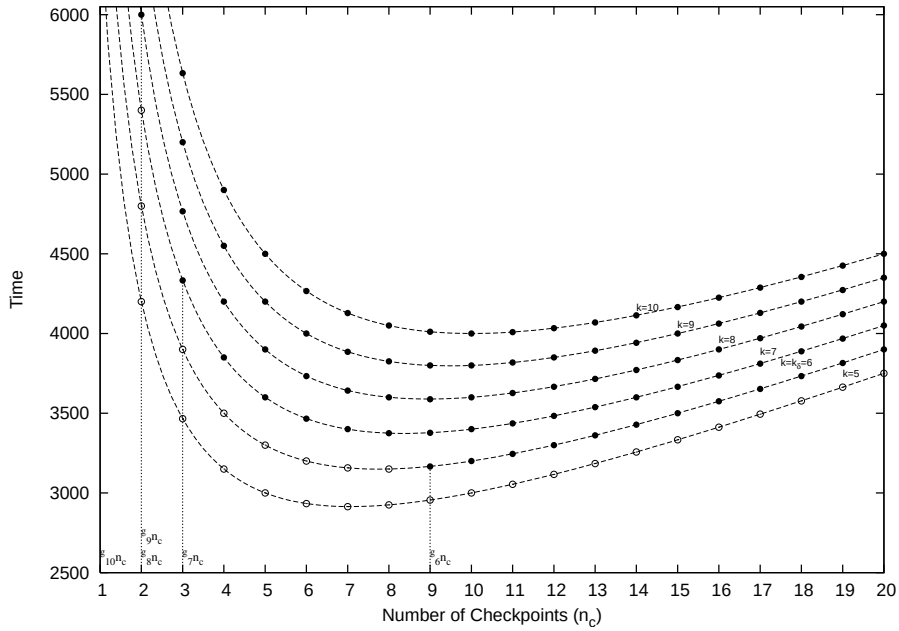


Figure 5.3.: Illustration of GCT_δ and ${}^{g}_{k}n_c$, for given $T = 1000$ t.u., $\tau = 110$ t.u., $P_T = 0.9$, $\delta = 0.99999998$, and $k \in [5, 10]$

an LoC requirement δ, the method finds the optimal number of checkpoints $n^*_{c_\delta}$ that results in the minimal GCT_δ.

A flow chart for the method is illustrated in Figure 5.4. As shown in Figure 5.4, the method consists of four different stages. Next, we discuss each stage in detail. In *Stage* I, the method obtains k_δ, *i.e.* the lowest integer value for the number of re-executions k that satisfies the following condition:

$$\bar{\lambda}_k \geq \delta \tag{5.21}$$

Finding k_δ is important because the necessary condition to obtain a GCT_δ is that the number or re-executions k has to be at least equal to k_δ. To obtain k_δ, as shown in Figure 5.4, first, k is initialized to zero and then, the condition in Eq. (5.21) is evaluated. If the condition in Eq. (5.21) is not satisfied, k is incremented and the condition is re-evaluated. Otherwise, if the condition in Eq. (5.21) is satisfied, then the current value of k represents k_δ. Once k_δ is obtained, the method proceeds with *Stage* II.

In *Stage* II, for the first time, the method identifies a GCT_δ. This GCT_δ is obtained by only exploring pairs of values $(k, {}_k n^*_c)$, where ${}_k n^*_c$ represents the

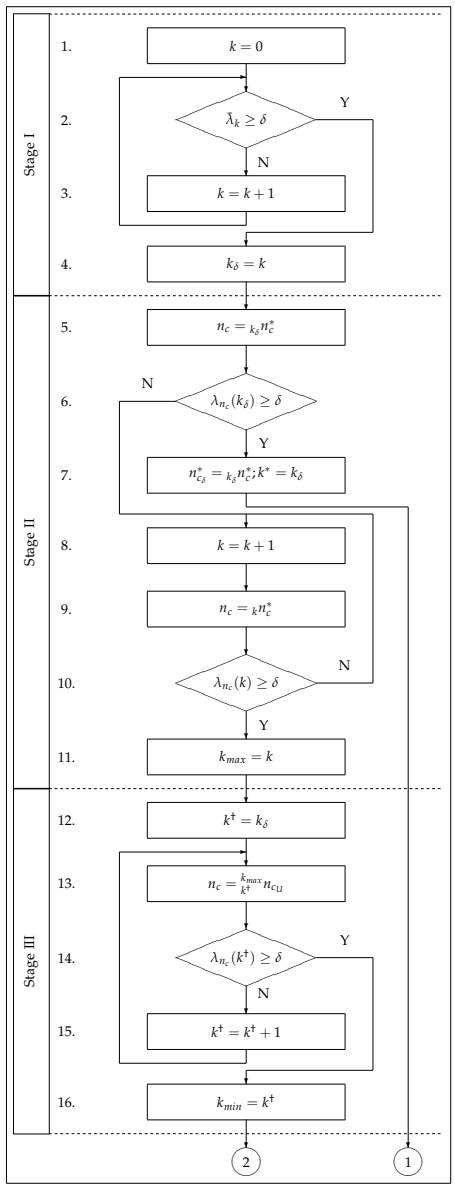


Figure 5.4.: Flow chart of presented method for minimizing GCT_δ

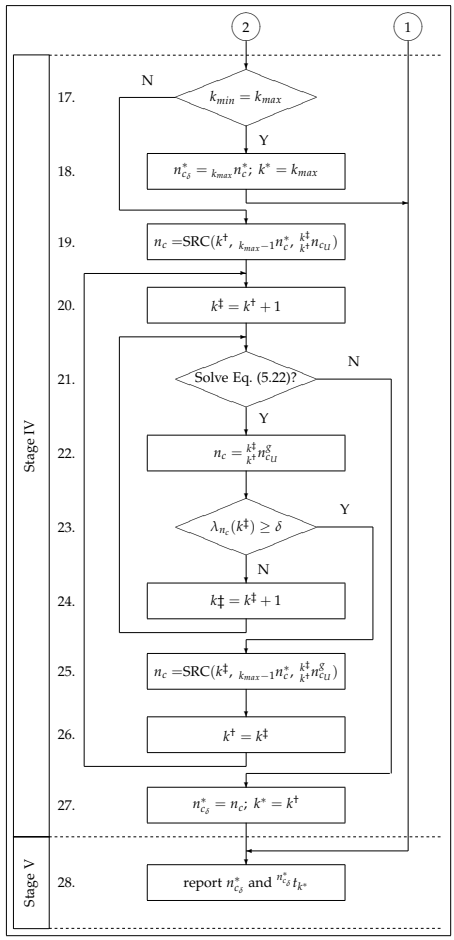


Figure 5.4.: Flow chart of presented method for minimizing GCT_δ (*continued*)

optimal number of checkpoints that provides the minimal completion time for the given k. The optimal number of checkpoints ${}_k n_c^*$ is calculated according to Eq. (5.9). Already at this stage, it may be possible to obtain the minimal GCT_δ and the optimal number of checkpoints $n_{c_\delta}^*$. Next, we elaborate the case when the minimal GCT_δ and the optimal number of checkpoints $n_{c_\delta}^*$ are obtained during *Stage* II.

As shown in Figure 5.4, the first step in *Stage* II is to calculate the optimal number of checkpoints ${}_k n_c^*$ that results in the minimal completion time for the given k (Eq. (5.9)). Since k is set to k_δ after completing *Stage* I, the first step in *Stage* II calculates the optimal number of checkpoints ${}_k n_c^* = {}_{k_\delta} n_c^*$. Next, the method checks whether the completion time that is obtained for the pair of values $(k_\delta, {}_{k_\delta} n_c^*)$ represents a GCT_δ. This is achieved by evaluating the condition given in Eq. (5.1). Only if the condition is satisfied, the method obtains the minimal GCT_δ, *i.e.* ${}^{k_\delta n_c^*} t_{k_\delta}$, and the optimal $n_{c_\delta}^* = {}_{k_\delta} n_c^*$ during *Stage* II. Observe that the minimal GCT_δ is obtained at this point because it represents the minimal completion time that is obtained for the lowest number of re-executions k_δ. As we already showed in the previous section, for a given k, there exist a minimal completion time that is achieved when ${}_k n_c^*$ checkpoints are used and we showed that by increasing k the minimal completion time ${}^{k n_c^*} t_k$ increases (observe the line that connects the optimal points of the curves illustrated in Figure 5.1). Since the necessary condition to obtain a GCT_δ is that the number of re-executions k has to be greater than or equal to k_δ, if already for $k = k_\delta$ the minimal completion time ${}^{k n_c^*} t_k$ represents a GCT_δ, then for any $k > k_\delta$ the minimal completion ${}^{k n_c^*} t_k$ will also represent a GCT_δ (this is consequence of **Theorem 2**). However, for any $k > k_\delta$, the minimal completion time ${}^{k n_c^*} t_k$ will be greater than ${}^{k_\delta n_c^*} t_{k_\delta}$. Therefore, it is possible to obtain the minimal GCT_δ already at *Stage* II and the only alternative that the method obtains the minimal GCT_δ at this stage is when the pair of values $(k_\delta, {}_{k_\delta} n_c^*)$ satisfy Eq. (5.1). In such case, the method proceeds with *Stage* V (see Figure 5.4).

While it may be possible to obtain the minimal GCT_δ during *Stage* II, this may not always be the case. For example, if the the pair of values $(k_\delta, {}_{k_\delta} n_c^*)$ does not satisfy Eq. (5.1), *Stage* II proceeds with the following steps. As shown in Figure 5.4, first, k is incremented and then, for the new value of k the optimal number of checkpoints ${}_k n_c^*$ is calculated using Eq. (5.9) and it is assigned to n_c. Next, the condition in Eq. (5.1) is evaluated for the pair of values (k, n_c). If the condition is not satisfied, then k is incremented, n_c is calculated for the latest value of k using Eq. (5.9) and the condition in Eq. (5.1) is re-evaluated. If the condition is satisfied, then the method proceeds with *Stage* III. Observe that before *Stage* III is reached, k has reached a value k_{max} such that the minimal completion time for this value represents a GCT_δ. Note that for any

value of k that is greater than k_{max} any GCT_δ will be greater than the GCT_δ achieved for the pair of values $(k_{max}, {}_{k_{max}}n_c^*)$. Therefore, the minimal GCT_δ will be obtained for a pair of values $(k, n_{c_\delta}^*)$ where $k \in [k_\delta, k_{max}]$.

In *Stage* III, the method identifies the lowest value for the number of re-executions that results in a GCT_δ that is lower than or equal to the GCT_δ identified in *Stage* II (${}_{k_{max}}{}^{n_c^*}t_{k_{max}}$). We denote this value with k_{min}. As discussed in the previous section, for any given $k^\dagger$ and $k^\ddagger$ such that $k^\dagger < k^\ddagger$ there exists a range of n_c values $[{}_{k^\dagger}^{k^\ddagger}n_{c_L}, {}_{k^\dagger}^{k^\ddagger}n_{c_U}]$ where for any n_c that belongs to the range the completion time ${}^{n_c}t_{k^\dagger}$ is lower than or equal to ${}^{n_c^*}_{k^\ddagger}t_{k^\ddagger}$. The boundaries of the range, *i.e.* ${}_{k^\dagger}^{k^\ddagger}n_{c_L}$ and ${}_{k^\dagger}^{k^\ddagger}n_{c_U}$, are calculated according to Eq. (5.19) and Eq. (5.20), respectively. The reason to obtain k_{min} is to further reduce the range of values for k, *i.e.* $k \in [k_{min}, k_{max}]$. The minimal GCT_δ will be obtained for a pair of values $(k, n_{c_\delta}^*)$ where $k \in [k_{min}, k_{max}]$.

To obtain k_{min}, a variable $k^\dagger$ is first initialized to k_δ (observe *Stage* III in Figure 5.4). Observe that k_{min} has to be greater or equal to k_δ in order to obtain a GCT_δ. Next, the upper bound ${}_{k^\dagger}^{k_{max}}n_{c_U}$ is calculated according to Eq. (5.20) and it is assigned to n_c. For the pair of values $(k^\dagger, n_c)$, the completion time ${}^{n_c}t_{k^\dagger}$ is lower than or equal to the GCT_δ that is obtained in *Stage* II (completion time for the pair of values $(k_{max}, {}_{k_{max}}n_c^*)$). However, the completion time ${}^{n_c}t_{k^\dagger}$ may not necessarily represent a GCT_δ. Therefore, the condition in Eq. (5.1) is evaluated. If the condition is not satisfied, then $k^\dagger$ is incremented, n_c is calculated according to Eq. (5.20) for the latest value of $k^\dagger$ and the condition is re-evaluated. If the condition is satisfied, k_{min} is evaluated as the latest value of the variable $k^\dagger$ and the method proceeds with *Stage* IV. Observe that k_{min} is either lower than or equal to k_{max}.

In *Stage* IV, the minimal GCT_δ is obtained. The minimal GCT_δ is either the GCT_δ obtained at *Stage* II, *i.e.* ${}_{k_{max}}{}^{n_c^*}t_{k_{max}}$, or a GCT_δ obtained for a pair of values $(k, {}_k^g n_c)$ where $k \in [k_{min}, k_{max})$. The notation ${}_k^g n_c$ represents the lowest value for the number of checkpoints such that for a given k and $n_c \geq {}_k^g n_c$, the pair of values (k, n_c) always results in a GCT_δ, *i.e.* the pair of values (k, n_c) satisfies Eq. (5.1).

As shown in Figure 5.4, the first step in *Stage* IV is to compare the variable $k^\dagger$ with k_{max}. Observe that the variable $k^\dagger$ is set to k_{min} after completing *Stage* III. If $k^\dagger = k_{max}$, it means that no GCT_δ lower than the GCT_δ obtained at *Stage* II has been identified and therefore, the minimal GCT_δ is the one obtained at *Stage* II, *i.e.* ${}_{k_{max}}{}^{n_c^*}t_{k_{max}}$. However, if the value of the variable $k^\dagger$, *i.e.* k_{min}, is lower than k_{max}, it means that for $k \in [k_{min}, k_{max})$ it is possible to obtain a GCT_δ lower than the GCT_δ obtained at *Stage* II. In such case, the minimal GCT_δ is obtained for a pair of values $(k, {}_k^g n_c)$.

As mentioned in the previous section, there is no closed-form expression to

calculate ${}^{g}_{k}n_c$ for a given k. However, for any $k \in [k_{min}, k_{max})$, ${}^{g}_{k}n_c$ is bounded and it belongs to the range $({}_{k_{max}-1}n^*_c, {}^{k_{max}}_{k_{min}}n_{c_U}]$. Observe that the lower bound is not included and therefore, ${}^{g}_{k}n_c$ has to be greater than ${}_{k_{max}-1}n^*_c$. This implies that all pairs of values $(k, {}_{k_{max}-1}n^*_c)$ do not satisfy Eq. (5.1). Next, we elaborate on how the lower and the upper bound of ${}^{g}_{k}n_c$ are obtained.

The lower bound of ${}^{g}_{k}n_c$, where $k \in [k_{min}, k_{max})$, is evaluated as ${}_{k_{max}-1}n^*_c$. Since in *Stage* II for all $k < k_{max}$ the pairs of values $(k, {}_{k}n^*_c)$ have not satisfied the condition in Eq. (5.1), the pair of values $(k_{max}-1, {}_{k_{max}-1}n^*_c)$ also does not result in a GCT_δ. If the pair of values $(k_{max}-1, {}_{k_{max}-1}n^*_c)$ does not satisfy Eq. (5.1), then according to **Theorem 2**, the pairs of values $(k, {}_{k_{max}-1}n^*_c)$ where $k < k_{max}$ also do not satisfy Eq. (5.1). Therefore, for any $k < k_{max}$, ${}^{g}_{k}n_c$ has to be greater than ${}_{k_{max}-1}n^*_c$.

The upper bound of ${}^{g}_{k}n_c$, where $k \in [k_{min}, k_{max})$, is evaluated as ${}^{k_{max}}_{k_{min}}n_{c_U}$. Observe that in *Stage* III, the method has identified k_{min}, which represents the lowest value for the number of re-executions that results in a GCT_δ that is lower than or equal to the GCT_δ identified in *Stage* II (${}_{k_{max}}{}^{n^*_c}t_{k_{max}}$). Since the pair of values $(k_{min}, {}^{k_{max}}_{k_{min}}n_{c_U})$ satisfies Eq. (5.1), *i.e.* it represents a GCT_δ, according to **Theorem 2**, the pairs of values $(k, {}^{k_{max}}_{k_{min}}n_{c_U})$, where $k \in [k_{min}, k_{max})$, will also satisfy Eq. (5.1). Therefore, for any $k \in [k_{min}, k_{max})$, ${}^{g}_{k}n_c$ has to be lower than or equal to ${}^{k_{max}}_{k_{min}}n_{c_U}$.

Although there is no closed-form expression to calculate ${}^{g}_{k}n_c$ for a given k, the fact that ${}^{g}_{k}n_c$ is bounded and it belongs to the range $({}_{k_{max}-1}n^*_c, {}^{k_{max}}_{k_{min}}n_{c_U}]$ for any $k \in [k_{min}, k_{max})$ allows us to find ${}^{g}_{k}n_c$ by using the search algorithm described in Figure 5.5. The basic idea of the search algorithm is to successively half the interval $(n_{c_L}, n_{c_U}]$ until an interval, where the difference between the end points is equal to one, is reached. Next, we detail the search algorithm given in Figure 5.5.

First, the search algorithm evaluates the difference $n_{c_U} - n_{c_L}$ and compares it with 1 (step 1 in Figure 5.5). If $n_{c_U} - n_{c_L} = 1$, then the algorithm returns n_{c_U}. Since only the upper bound of the interval $(n_{c_L}, n_{c_U}]$ satisfies Eq. (5.1), ${}^{g}_{k}n_c$ is evaluated as n_{c_U}. However, if the difference $n_{c_U} - n_{c_L}$ is not equal to one, the algorithm proceeds with step 4. In step 4, the the median n_{c_m} of the current interval is evaluated, *i.e.* $n_{c_m} = \lfloor (n_{c_L} + n_{c_U})/2 \rfloor$. Next, the search algorithm checks whether the pair of values (k, n_{c_m}) satisfies Eq. (5.1) (step 5 in Figure 5.5). If the pair of values (k, n_{c_m}) satisfies Eq. (5.1), then the algorithm steps into a recursive call, where the upper bound of the interval n_{c_U} is updated and set to n_{c_m} (step 6 in Figure 5.5). However, if the pair of values (k, n_{c_m}) does not satisfy Eq. (5.1), the algorithm steps into a recursive call, where the lower bound of the interval n_{c_L} is updated and set to n_{c_m}, and it

ALGORITHM SRC(k, n_{c_L}, n_{c_U})

1 **.if** $(n_{c_U} - n_{c_L} = 1)$ **then**
2 . **return** n_{c_U}
3 **.else**
4 . $n_{c_m} = \lfloor (n_{c_L} + n_{c_U})/2 \rfloor$;
5 . **if** $(\lambda_{n_{c_m}}(k) < \delta)$ **then**
6 . **return** SRC(k, n_{c_m}, n_{c_U})
7 . **else**
8 . **return** SRC(k, n_{c_L}, n_{c_m})

Figure 5.5.: Search algorithm for obtaining ${}^{g}_{k}n_c$

keeps on searching ${}^{g}_{k}n_c$ in the interval $(n_{c_m}, n_{c_U}]$ (step 8 in Figure 5.5). By using this algorithm, ${}^{g}_{k}n_c$ is obtained and this is important as the minimal GCT_δ in *Stage* IV may be evaluated for a pair of values $(k, {}^{g}_{k}n_c)$ where $k \in [k_{min}, k_{max})$.

We mentioned earlier that the first step in *Stage* IV is to compare $k^\dagger$ with k_{max} and we concluded that only when the value of $k^\dagger$, *i.e.* k_{min}, is lower than k_{max}, the minimal GCT_δ will be obtained for a pair of values $(k, {}^{g}_{k}n_c)$ where $k \in [k_{min}, k_{max})$. Thus, if $k^\dagger < k_{max}$, the method identifies ${}^{g}_{k^\dagger}n_c$ by using the search algorithm given in Figure 5.5 (see the second step in *Stage* IV in Figure 5.4). To obtain ${}^{g}_{k^\dagger}n_c$, the search algorithm is invoked with the following arguments: $k^\dagger$, ${}_{k_{max}-1}n^*_c$ and ${}^{k_{max}}_{k_{min}}n_{c_U}$. Observe that the GCT_δ identified for the pair of values $(k^\dagger, {}^{g}_{k^\dagger}n_c)$ is lower than the GCT_δ obtained during *Stage* II and therefore, it is a candidate for the minimal GCT_δ. However, it is not guaranteed that the current pair of values $(k^\dagger, {}^{g}_{k^\dagger}n_c)$ will provide the minimal GCT_δ since it may be possible that the minimal GCT_δ is achieved for another pair of values $(k^\ddagger, {}^{g}_{k^\ddagger}n_c)$ where $k^\dagger < k^\ddagger < k_{max}$. Therefore, the method proceeds with the following steps. First, $k^\ddagger$ is set to $k^\dagger + 1$. For $k^\ddagger$, it may be possible (but not always) to find a range of values for n_c, denoted with $[{}^{k^\ddagger}_{k^\dagger}n^g_{c_L}, {}^{k^\dagger}_{k^\dagger}n^g_{c_U}]$, such that for any n_c that belongs to this range, the completion time obtained for the pair of values $(k^\ddagger, n_c)$ is lower than or equal to the completion time obtained for the pair of values $(k^\dagger, {}^{g}_{k^\dagger}n_c)$. This is expressed with Eq. (5.22).

$$
\begin{aligned}
& n_c t_{k^\ddagger} \leq {}^{g}_{k^\dagger}n_c \, t_{k^\dagger} \\
& \Rightarrow \tau \times {n_c}^2 + \underbrace{\left(k^\ddagger \times \tau - \left({}^{g}_{k^\dagger}n_c \times \tau + k^\dagger \times \left(\frac{T}{{}^{g}_{k^\dagger}n_c} + \tau \right) \right) \right)}_{D} \times n_c + k^\ddagger \times T \leq 0 \\
& \Rightarrow \tau \times {n_c}^2 + D \times n_c + k^\ddagger \times T \leq 0 \qquad (5.22)
\end{aligned}
$$

Eq. (5.22) is similar to Eq. (5.13). However, while the roots of Eq. (5.13) are always evaluated as real numbers, the roots of Eq. (5.22) may not always be evaluated as real numbers. Based on the discussion regarding Eq. (5.13) in the previous section, if the roots of Eq. (5.22) are evaluated as complex numbers, it means that Eq. (5.22) will not be satisfied for any n_c value. In such case, any GCT_δ obtained for any $k \geq k^\ddagger$ will be larger than the GCT_δ obtained for the pair of values $(k^\dagger, {}^{g}_{k^\dagger}n_c)$ and therefore, the minimal GCT_δ will be the one obtained for the pair of values $(k^\dagger, {}^{g}_{k^\dagger}n_c)$ (observe the "N" branch of the decision box "Solve Eq. (5.22)" shown in *Stage* IV in Figure 5.4). However, if the roots of Eq. (5.22), *i.e.* ${}^{k^\ddagger}_{k^\dagger}n^{g}_{c_L}$ and ${}^{k^\ddagger}_{k^\dagger}n^{g}_{c_U}$, are evaluated as real numbers, that means that Eq. (5.22) will be satisfied for any $n_c \in [{}^{k^\ddagger}_{k^\dagger}n^{g}_{c_L}, {}^{k^\ddagger}_{k^\dagger}n^{g}_{c_U}]$. The roots of Eq. (5.22), *i.e.* ${}^{k^\ddagger}_{k^\dagger}n^{g}_{c_L}$ and ${}^{k^\ddagger}_{k^\dagger}n^{g}_{c_U}$ are presented in Eq. (5.23) and Eq. (5.24), respectively.

$$
{}^{k^\ddagger}_{k^\dagger}n^{g}_{c_L} = \left\lceil \frac{-D - \sqrt{D^2 - 4 \times \tau \times k^\ddagger \times T}}{2 \times \tau} \right\rceil \qquad (5.23)
$$

$$
{}^{k^\ddagger}_{k^\dagger}n^{g}_{c_U} = \left\lfloor \frac{-D + \sqrt{D^2 - 4 \times \tau \times k^\ddagger \times T}}{2 \times \tau} \right\rfloor \qquad (5.24)
$$

$$
(5.25)
$$

The constant D used in Eq. (5.23) and Eq. (5.24) is evaluated with the following expression:

$$
D = k^\ddagger \times \tau - \left({}^{g}_{k^\dagger}n_c \times \tau + k^\dagger \times \left(\frac{T}{{}^{g}_{k^\dagger}n_c} + \tau \right) \right) \qquad (5.26)
$$

Thus, for a given $k^\ddagger$, if the roots of Eq. (5.22) are evaluated as real numbers, the completion time obtained for the pair of values $(k^\ddagger, {}^{k^\ddagger}_{k^\dagger}n^{g}_{c_U})$, where ${}^{k^\ddagger}_{k^\dagger}n^{g}_{c_U}$ is calculated according to Eq. (5.24), will be lower than or equal to the GCT_δ obtained for the pair of values $(k^\dagger, {}^{g}_{k^\dagger}n_c)$. However, the completion time

obtained for the pair of values $(k^{\ddagger}, {}^{k^{\ddagger}}_{k^{\dagger}}n^{g}_{c_U})$, where ${}^{k^{\ddagger}}_{k^{\dagger}}n^{g}_{c_U}$ is calculated according to Eq. (5.24), does not necessarily need to represent a GCT_{δ}. Therefore, in the next step, the condition in Eq. (5.1) is evaluated for the pair of values $(k^{\ddagger}, {}^{k^{\ddagger}}_{k^{\dagger}}n^{g}_{c_U})$ (see the decision box "$\lambda_{n_c}(k^{\ddagger}) \geq \delta$" in *Stage* IV in Figure 5.4). If the pair of values $(k^{\ddagger}, {}^{k^{\ddagger}}_{k^{\dagger}}n^{g}_{c_U})$ does not satisfy Eq. (5.1), $k^{\ddagger}$ is incremented and Eq. (5.22) is solved for the latest value of $k^{\ddagger}$. These two steps, *i.e.* incrementing $k^{\ddagger}$ and solving Eq. (5.22), are repeated until $k^{\ddagger}$ has reached a value such that either (1) the roots of Eq. (5.22) are evaluated as complex numbers, in which case the minimal GCT_{δ} is obtained for the pair of values $(k^{\dagger}, {}^{g}_{k^{\dagger}}n_c)$ or (2) the pair of values $(k^{\ddagger}, {}^{k^{\ddagger}}_{k^{\dagger}}n^{g}_{c_U})$ represents a GCT_{δ}.

If the pair of values $(k^{\ddagger}, {}^{k^{\ddagger}}_{k^{\dagger}}n^{g}_{c_U})$ represents a GCT_{δ}, it means that even a lower GCT_{δ} can be obtained for the pair of values $(k^{\ddagger}, {}^{g}_{k^{\ddagger}}n_c)$. To find ${}^{g}_{k^{\ddagger}}n_c$, the search algorithm given in Figure 5.5 is invoked with the following arguments: $k^{\ddagger}$, ${}_{k_{max}-1}n^{*}_{c}$ and ${}^{k^{\ddagger}}_{k^{\dagger}}n^{g}_{c_U}$. Next, the value of $k^{\ddagger}$ is assigned to $k^{\dagger}$. By this, it is guaranteed that $k^{\dagger}$ always points to a value for which the lowest GCT_{δ} is identified. Next, the method proceeds by checking if it is possible to find another $k^{\ddagger}$ greater than $k^{\dagger}$ such that it results in even lower GCT_{δ}. If for any $k^{\ddagger} > k^{\dagger}$ it is not possible to achieve lower GCT_{δ} than the GCT_{δ} obtained for the pair of values $(k^{\dagger}, {}^{g}_{k^{\dagger}}n_c)$, then the minimal GCT_{δ} is obtained for the pair of values $(k^{\dagger}, {}^{g}_{k^{\dagger}}n_c)$, and the optimal number of checkpoints is evaluated as ${}^{g}_{k^{\dagger}}n_c$.

Finally, in *Stage* V, the optimal number of checkpoints $n^{*}_{c_{\delta}}$ and the minimal GCT_{δ} are reported.

To clarify how the method finds the optimal number of checkpoints $n^{*}_{c_{\delta}}$ and the minimal GCT_{δ}, in the following section, we present a detailed example.

5.3.1. EXAMPLE

In this section, we illustrate the different pairs of values (k, n_c) that are explored by the method, discussed in the previous section, for the following example. Important to note is that the example was designed such that all different paths in the flow chart presented in Figure 5.4 are exercised.

Example 1. Given the following parameters:

- a processing time $T = 1000$ t.u.,
- a checkpointing overhead $\tau = 200$ t.u.,
- a probability $P_T = 0.5$ that no errors occur within an interval of length T, and
- an LoC requirement $\delta = 1 - 10^{-18}$

find the minimal GCT_{δ} and the optimal number of checkpoints $n^{*}_{c_{\delta}}$.

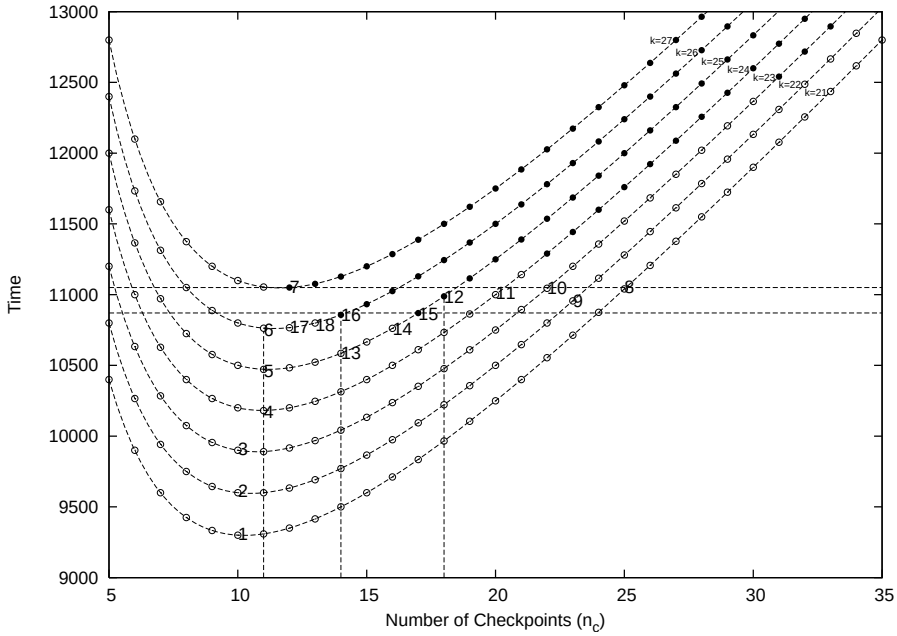


Figure 5.6.: Illustration on finding the minimal GCT_δ, for given $T = 1000$ t.u., $\tau = 20$ t.u., $P_T = 0.5$, and $\delta = 1 - 10^{-18}$

For the given example, we illustrate in Figure 5.6 the different pairs of values (k, n_c) that are explored during the different stages of the method that finds the minimal GCT_δ, and the optimal number of checkpoints $n^*_{c_\delta}$.

After completing *Stage* I (see Figure 5.4), the method obtains the lowest value for the number of re-executions for which a GCT_δ can be obtained, *i.e.* $k_\delta = 21$. Therefore, in Figure 5.6, we only plot the completion time for $k \geq 21$. Note that the empty circles shown in Figure 5.6 represent instances of the completion time which do not represent a GCT_δ while the filled circles represent a GCT_δ .

During *Stage* II (see Figure 5.4), the method explores the pairs of values $(k, {}_k n^*_c)$, where ${}_k n^*_c$ is calculated according to Eq. (5.9), until it finds the first pair of values $(k_{max}, {}_{k_{max}} n^*_c)$ which satisfies Eq. (5.1). The pairs of values $(k, {}_k n_c)$ that are explored during *Stage* II are illustrated with the points marked with "1"–"7" in Figure 5.6 (observe that the points represent the minimum of each of the different curves for $21 \leq k \leq 27$ in the figure). As mentioned in the previous section, if and only if the pair of values $(k_\delta, {}_{k_\delta} n^*_c)$ represents a GCT_δ, the minimal GCT_δ is obtained in *Stage* II. However, for the given example, this

is not the case. As shown in Figure 5.6, the pair of values $(k_\delta = 21, {}_{k_\delta}n_c^* = 10)$, marked with point "1", does not represent a GCT_δ (observe the empty circle next to "1" in the figure). If the minimal GCT_δ is not obtained during *Stage* II, this stage completes when a GCT_δ is identified for the pair of values $(k_{max}, {}_{k_{max}}n_c^*)$. Observe that the points marked with "1"-"6" in Figure 5.6 do not represent a GCT_δ (observe the empty circle next to each of these points in the figure) and only the point marked with "7", *i.e.* the pair of values $(k = 27, {}_k n_c^* = 12)$, represents a GCT_δ (observe the filled circle next to the point marked with "7" in the figure). Thus, after completing *Stage* II, for this example, the method identifies $k_{max} = 27$.

In *Stage* III, the method identifies the lowest value k_{min} for the number of re-executions that results in a GCT_δ lower than or equal to the GCT_δ identified in *Stage* II. As stated earlier, for the given example, the GCT_δ in *Stage* II was obtained for the pair of values $(k = 27, {}_k n_c^* = 12)$ and it is illustrated with the point marked with "7" in Figure 5.6. The pairs of values $(k, {}_k n_c)$ that are explored during *Stage* III are illustrated with the points marked with "8"–"12" in Figure 5.6. Observe that these points are instances of the completion time for $21 \leq k \leq 27$ that lie just below the horizontal line that passes through the point marked with "7". The points marked with "8"–"12" in Figure 5.6 represent the pair of values $(k, {}_k^{k_{max}} n_{c_U})$, where ${}_k^{k_{max}} n_{c_U}$ is calculated according to Eq. (5.20). The first step in *Stage* III explores the pair of values $(k = 21, {}_k^{k_{max}} n_{c_U} = 25)$ and it is illustrated with the point marked with "8" in Figure 5.6. As shown in the figure, the point marked with "8" does not represent a GCT_δ (observe the empty circle next to the "8"), and therefore k_{min} is not yet evaluated. In the next step, the method explores the pair of values $(k = 22, {}_k^{k_{max}} n_{c_U} = 23)$ which is illustrated with the point marked with "9" in Figure 5.6. Again, this point does not represent a GCT_δ. Finally, when the pair of values $(k = 25, {}_k^{k_{max}} n_{c_U} = 18)$, illustrated with "12" in Figure 5.6, is explored, a GCT_δ is identified. Therefore, $k_{min} = 25$ is identified at the end of *Stage* III.

The minimal GCT_δ is obtained during *Stage* IV (see Figure 5.4). Since k_{min} and k_{max} are not the same, *i.e.* $k_{min} = 25$ (identified during *Stage* III) and $k_{max} = 27$ (identified during *Stage* II), the minimal GCT_δ is obtained for a pair of values $(k, {}_k^g n_c)$ where $25 \leq k < 27$. The first step during *Stage* IV is to find ${}_k^g n_c$ for $k = 25$. This is done by using the search algorithm described in Figure 5.5. To obtain ${}_k^g n_c$ for $k = 25$, the search algorithm requires the lower and the upper bound of ${}_k^g n_c$. As discussed in the previous section, the lower bound of ${}_k^g n_c$ is evaluated as ${}_{k_{max}-1}n_c^*$ and it represents the number of checkpoints for which the minimum of the curve $k = k_{max} - 1 = 26$ is obtained. In Figure 5.6, this minimum is illustrated with the point marked with "6", *i.e.* the pair of values $(k_{max} - 1 = 26, {}_{k_{max}-1}n_c^* = 11)$. The lower bound of ${}_k^g n_c$ for $k = 25$ is

illustrated with the vertical line that passes through the point marked with "6", *i.e.* $n_c = 11$. The upper bound of ${}^{g}_{k}n_c$ is evaluated as ${}^{k_{max}}_{k_{min}}n_{c_U}$, and it represents the number of checkpoints for the intersection point between the curve representing the completion time for $k = k_{min} = 25$ and the completion time obtained for the pair of values $(k_{max} = 27, {}_{k_{max}}n^*_c = 12)$, illustrated by the horizontal line that passes through the point marked with "7" in Figure 5.6. The intersection point is illustrated with "12" and it represents the pair of values $(k = 25, {}^{k_{max}}_{k}n_{c_U} = 18)$. Thus, the upper bound of ${}^{g}_{k}n_c$ for $k = 25$ is illustrated with the vertical line that passes through the point marked with "12", *i.e.* $n_c = 18$. Given that $k = 25$, $n_{c_L} = 11$, and $n_{c_U} = 18$, the search algorithm presented in Figure 5.5 obtains the ${}^{g}_{k}n_c$ while exploring the following pairs of values (k, n_c). Since the difference $n_{c_U} - n_{c_L}$ is not equal to one, first, the search method obtains $n_{c_m} = \lfloor (11 + 18)/2 \rfloor = 14$, and then the pair of values $(k = 25, n_c = 14)$ illustrated with the point marked with "13" in Figure 5.6 is explored. As shown in Figure 5.6, the point marked with "13" does not represent a GCT_δ. Therefore, the search algorithm proceeds by exploring the interval $(n_{c_L} = 14, n_{c_U} = 18]$. Since again the difference $n_{c_U} - n_{c_L}$ is not equal to one, new $n_{c_m} = \lfloor (14 + 18)/2 \rfloor = 16$ is calculated, and then the pair of values $(k = 25, n_c = 16)$ illustrated with the point marked with "14" in Figure 5.6 is explored. Observe from Figure 5.6 that the point marked with "14" does not represent a GCT_δ. Therefore, the search algorithm proceeds by exploring the interval $(n_{c_L} = 16, n_{c_U} = 18]$. Once again, the difference $n_{c_U} - n_{c_L}$ is not equal to one. Therefore, $n_{c_m} = \lfloor (16 + 18)/2 \rfloor = 17$ is calculated, and next the pair of values $(k = 25, n_c = 17)$ illustrated with the point marked with "15" in Figure 5.6 is explored. Observe that "15" represents a GCT_δ. This forces the search to proceed in the interval $(n_{c_L} = 16, n_{c_U} = 17]$. However, since the difference $n_{c_U} - n_{c_L}$ is equal to one, the search algorithm identifies ${}^{g}_{k}n_c = 17$ for $k = k_{min} = 25$. Once the search algorithm has identified ${}^{g}_{k}n_c = 17$ for $k = k_{min} = 25$, the pair of values $(k_{min} = 25, {}^{g}_{k_{min}}n_c = 17)$, illustrated with "15" in Figure 5.6, provides the lowest GCT_δ that has been obtained until this point. However, *Stage* IV does not end here (see Figure 5.4).

The next step after finding ${}^{g}_{k}n_c$ for $k = k_{min} = 25$ in *Stage* IV, is to check if it is possible to obtain a lower GCT_δ for $k = 26$. Therefore, the method solves Eq. (5.22). The roots of Eq. (5.22) are evaluated as real numbers. In Figure 5.6, this is shown with the intersection of the horizontal line that passes through the point marked with "15" (the completion time for the pair of values $(k_{min} = 25, {}^{g}_{k_{min}}n_c = 17)$) and the curve which represents the completion time for $k = 26$. Using Eq. (5.24), the method first obtains ${}^{k^\ddagger=26}_{k^\dagger=25}n^g_{c_U} = 14$, and next it checks whether the pair of values $(k^\ddagger = 26, {}^{k^\ddagger=26}_{k^\dagger=25}n^g_{c_U} = 14)$ illustrated with "16" in Figure 5.6 represents a GCT_δ. As shown in in Figure 5.6, the

point marked with "16" represents a GCT_δ. Therefore, the method proceeds by finding ${}^{g}_{k}n_c$ for $k = 26$. To obtain ${}^{g}_{k}n_c$ for $k = 26$ the search algorithm is invoked with the following arguments: $k = 26$, $n_{c_L} = 11$, and $n_{c_U} = 14$. The lower and the upper bound of ${}^{g}_{k}n_c$ for $k = 26$ are shown with the vertical lines that pass through the points marked with "6" and "16", respectively. Since for the given $n_{c_L} = 11$ and $n_{c_U} = 14$ the difference $n_{c_U} - n_{c_L}$ is not equal to one, the search method first calculates $n_{c_m} = \lfloor (11 + 14)/2 \rfloor = 12$, and next it explores the pair of values $(k = 26, n_c = 12)$, which is illustrated with "17" in Figure 5.6. As shown in Figure 5.6, the point "17" does not represent a GCT_δ. Therefore, the search proceeds in the interval $(n_{c_L} = 12, n_{c_U} = 14]$. Again, since the difference $n_{c_U} - n_{c_L}$ is not equal to one, the search method calculates $n_{c_m} = \lfloor (12 + 14)/2 \rfloor = 13$, and next it explores the pair of values $(k = 26, n_c = 13)$, which is illustrated with "18" in Figure 5.6. As shown in Figure 5.6, the point "18" does not represent a GCT_δ, and therefore the search proceeds in the interval $(n_{c_L} = 13, n_{c_U} = 14]$. However, this time, the difference $n_{c_U} - n_{c_L}$ is equal to one, and therefore the search method finds ${}^{g}_{k}n_c = 14$ for $k = 26$. The pair of values $(k = 26, {}^{g}_{k}n_c = 14)$, illustrated with "16" in Figure 5.6, represents the minimal GCT_δ for the given example.

5.4. EXPERIMENTAL RESULTS

The purpose of the experiments presented in this section is to: (1) show that GCT_δ varies with the number of checkpoints and (2) validate the proposed method that finds the minimal GCT_δ and the optimal number of checkpoints $n^*_{c_\delta}$. For that reason, we present results for the following experiment:

- P1: find the optimal number of checkpoints $n^*_{c_\delta}$ that minimizes the GCT_δ, for a given LoC requirement δ.

For the given experiment, we use two input scenarios, Scenario A and Scenario B, which are summarized in Table 5.1. For each scenario, the following inputs are given: a processing time T, a checkpointing overhead τ, and a probability P_T that no errors occur in a processor within an interval equal to T. Furthermore, we assume given is an LoC requirement $\delta = 1 - 10^{-10}$.

The results for P1 are presented in Table 5.2 and Table 5.3 for Scenario A and Scenario B, respectively. The results show, for different number of checkpoints n_c, the minimum number of re-executions k that are needed to provide a GCT_δ, along with the obtained GCT_δ. The results are obtained as follows. For each n_c, we initialize $k = 1$ and check the following inequality $\lambda_{n_c}(k) \geq \delta$. If the inequality is not satisfied, then k is incremented until $\Lambda_{n_c}(t_k) \geq \delta$. In other words, for each n_c, we find the lowest instance of the completion time that represents a GCT_δ. In Table 5.2 and Table 5.3, for each n_c, we present

Scenario A	Scenario B
T = 1000 t.u.	T = 1000 t.u.
τ = 20 t.u.	τ = 20 t.u.
P_T = 0.99999	P_T = 0.9

Table 5.1.: Input scenarios

$\delta = 1 - 10^{-10}$					
n_c	k	GCT_δ	n_c	k	GCT_δ
1	2	3060	11	2	1442
2	2	2080	12	2	1447
3	2	1767	13	2	1454
4	2	1620	14	2	1463
5	2	1540	15	2	1474
6	2	1494	16	2	1485
7	2	1466	17	2	1498
8	2	1450	18	2	1512
9	2	1443	19	2	1526
10	**2**	**1440**	20	2	1540

Table 5.2.: GCT_δ and the number of re-executions k included in GCT_δ, for Scenario A, at various number of checkpoints n_c

(1) the minimum required number of re-executions k to obtain a GCT_δ and (2) the obtained GCT_δ .

Table 5.2 shows the results for Scenario A. For the given $P_T = 0.99999$ in Scenario A, the given LoC requirement $\delta = 1 - 10^{-10}$ can be reached for $k_\delta = 2$. Using **Theorem 4** we evaluate for $k \geq 0$ the limit $\bar{\lambda}_k$ and compare it with δ. For $k = 0$ and $k = 1$ the limit $\bar{\lambda}_k$ is lower than δ. For $k = 2$ we obtain a limit $\bar{\lambda}_k$ larger than δ which leads to $k_\delta = 2$. As can be seen from Table 5.2, already for $n_c = 1$ a GCT_δ is obtained for $k = k_\delta = 2$. As a consequence of **Theorem 3**, all pairs (n_c, k_δ) will represent a GCT_δ (observe that the value of k is equal to 2 for all n_c in Table 5.2). We have already discussed in Section 5.2 that for a fixed $k \geq 0$ there exists an optimal number of checkpoints ${}_k n_c^*$ such that the pair of values $(k, {}_k n_c^*)$ results in the minimal completion time for the given k. The optimal ${}_k n_c^*$ is evaluated with Eq. (5.9). By using the inputs given

$\delta = 1 - 10^{-10}$					
n_c	k	GCT_δ	n_c	k	GCT_δ
1	13	14280	12	8	2066
2	11	6760	13	8	2036
3	10	4594	14	8	2012
4	9	3510	15	8	1994
5	9	3080	16	8	1980
6	9	2800	17	8	1971
7	8	2443	18	8	1965
8	8	2320	19	8	1962
9	8	2229	**20**	**8**	**1960**
10	8	2160	21	8	1961
11	8	2108	22	8	1964

Table 5.3.: GCT_δ and the number of re-executions k included in GCT_δ, for Scenario B, at various number of checkpoints n_c

for Scenario A in Eq. (5.9), we get:

$$\begin{aligned} n_{c_L} &= \left\lfloor \sqrt{2 \times \tfrac{1000}{20}} \right\rfloor = 10 \\ n_{c_U} &= \left\lceil \sqrt{k \times \tfrac{1000}{20}} \right\rceil = 10 \end{aligned} \tag{5.27}$$

As $n_{c_L} = n_{c_U}$, the optimal ${}_k n_c^* = n_{c_L} = n_{c_U} = 10$. By observing the results in Table 5.2, we see that as n_c increases from 1 to 10, the GCT_δ decreases. However, increasing n_c above 10 results in a higher GCT_δ. Thus, the minimal GCT_δ is reached at $n_{c_\delta}^* = 10$. The minimal GCT_δ includes two re-executions, and it is evaluated as $GCT_\delta = 1000 + 10 \times 20 + 2 \times (\frac{1000}{10} + 20) = 1440$t.u. The same results are produced when the method discussed in Section 5.3 is used. In *Stage* I (see Figure 5.4), the method computes $k_\delta = 2$ for the given inputs in Scenario A. Already at *Stage* II, the method obtains the minimal GCT_δ. In *Stage* II, the method first calculates ${}_k n_c^*$ using Eq. (5.9) for $k = k_\delta = 2$ and it evaluates ${}_{k_\delta} n_c^* = 10$. Since the pair of values $(k_\delta = 2, {}_{k_\delta} n_c^* = 10)$ satisfies Eq. (5.1), the method outputs the minimal $GCT_\delta = {}^{10}t_2 = 1440$ t.u. and the optimal number of checkpoints $n_{c_\delta}^* = 10$. The advantage of the method is that it obtains the minimal GCT_δ and the optimal $n_{c_\delta}^*$ in much less iterations. For Scenario A, the minimal GCT_δ reported in Table 5.2 is achieved after 10 iterations, while the method obtains the minimal GCT_δ in a single iteration (only the pair of values $(k_\delta = 2, {}_{k_\delta} n_c^* = 10)$ is explored).

Table 5.3 shows the results for Scenario B. For the given $P_T = 0.9$ in Scenario B, the given LoC requirement $\delta = 1 - 10^{-10}$ can be reached for $k_\delta = 8$. Observe in Table 5.3 that only when $n_c \geq 7$ a GCT_δ can be achieved for $k = k_\delta = 8$. Thus, for all $n_c \geq 7$ all pairs (n_c, k_δ) will represent a GCT_δ. That means that ${}^{g}_{k_\delta} n_c = 7$. Observe that for all $n_c < {}^{g}_{k_\delta} n_c = 7$ a GCT_δ can only be obtained for a $k > k_\delta = 8$, *e.g.* for $n_c = 3$ a GCT_δ is achieved for $k = 10 > k_\delta = 8$. For the fixed $k = k_\delta = 8$ and the inputs given in Scenario B, the optimal ${}_k n_c^*$ is evaluated as 20 (using Eq. (5.9)). Since ${}^{g}_{k_\delta} n_c = 7 < {}_{k_\delta} n_c^* = 20$, as discussed in Section 5.2, the minimal GCT_δ will be obtained for the pair of values $(k_\delta, {}_{k_\delta} n_c^*)$. By observing the results in Table 5.3 we see that as n_c increases from 1 to 20, the GCT_δ decreases. However, increasing n_c above 20 results in a higher GCT_δ. Thus, the minimal GCT_δ is reached at $n_{c_\delta}^* = 20$ and it is evaluated as $GCT_\delta = 1000 + 20 \times 20 + 8 \times (\frac{1000}{20} + 20) = 1000 + 400 + 8 \times 70 = 1960$ t.u. The same results are obtained when the proposed method, discussed in Section 5.3, is used. Similar to Scenario A, the method obtains the minimal GCT_δ for Scenario B during *Stage* II, and therefore it obtains the minimal GCT_δ in a single iteration.

From the results, we conclude that GCT_δ varies with the number of checkpoints n_c, and there exists an optimal number of checkpoints $n_{c_\delta}^*$ that results in the minimal GCT_δ. To obtain the optimal number of checkpoints $n_{c_\delta}^*$ that results in the minimal GCT_δ, we proposed a method in Section 5.3. As demonstrated with the experiments in this section, the proposed method always identifies the minimal GCT_δ and the optimal number of checkpoints $n_{c_\delta}^*$. The advantage of the proposed method is that $n_{c_\delta}^*$ and the minimal GCT_δ can be identified in much less iterations in comparison to finding for each number of checkpoints n_c the lowest instance of the completion time that represents a GCT_δ.

6

Summary of Part I

This chapter presents a summary of Part I where we discussed optimization of RRC for three different optimization objectives while assuming equidistant checkpointing. The optimization objectives discussed in Part I are: (1) *Average Execution Time*, (2) *Level of Confidence* and (3) *Guaranteed Completion Time*. Next, we outline the contributions with respect to each of the optimization objectives stated earlier.

For the optimization of *Average Execution Time*, we analyzed the impact of the number of checkpoints on the execution time when RRC is used. We developed a mathematical framework to compute the average execution time (AET) for a given number of checkpoints. Furthermore, we derived a closed-form mathematical expression to compute the optimal number of checkpoints that results in the minimal AET. These results are of high importance for soft real-time systems (RTSs) where minimizing the AET is one of the major optimization objectives.

For the optimization of *Level of Confidence*, we analyzed the impact of the number of checkpoints on the probability to meet a given deadline. While AET is mainly applicable for soft RTSs, the Level of Confidence (LoC) is equally applicable for both soft and hard RTSs as it provides the probability to meet a given deadline. The analyses were conducted for the following two cases: (1) a single job constrained with a deadline and (2) multiple jobs constrained with a global deadline.

For the single job case, we derived a mathematical expression to evaluate the LoC with respect to a given deadline, *i.e.* the probability to meet a given deadline, for a given number of checkpoints. The expression was thoroughly analyzed and important properties of the expression along with proofs were provided. Using these properties, we provided directions on how to identify the optimal number of checkpoints that maximizes the LoC. The optimal number of checkpoints is identified by only evaluating and comparing the LoC for a specific set of values for the number of checkpoints. These values

for the number of checkpoints are obtained by using a closed-form mathematical expression that was derived by using the properties of the expression that is used to calculate the LoC.

For the multiple jobs case, we focused on optimizing the LoC with respect to a given global deadline for a set of jobs. We showed that directly applying the results obtained from optimization of the LoC for each individual job does not provide the optimal solution for the case of multiple jobs. In particular, we investigated two approaches: (1) Local Optimization approach, where the number of checkpoints assigned to each job was obtained by optimizing the LoC for a single job with respect to a local deadline; and (2) Single Large Job approach, where the set of jobs was considered as one job. Both approaches fail to provide the optimal solution, *i.e.* a checkpoint assignment (a vector where each element defines the number of checkpoints to be used for each individual job) that results in the maximal LoC with respect to the global deadline. We derived a mathematical expression to compute the LoC with respect to the given global deadline for a given checkpoint assignment. As one solution to obtain the optimal checkpoint assignment is to explore all possible checkpoint assignments, an exhaustive search approach was analyzed. The exhaustive search approach is important because it always finds the optimal checkpoint assignment. Due to the high problem complexity and the fact that the exhaustive search approach is very time-consuming, we developed a method that speeds up the computations and obtains the solution in much shorter time. The proposed method was compared against the exhaustive search method and in all the experiments, the proposed method was able to find the optimal checkpoint assignment.

Important to note is that the mathematical framework developed for evaluation of the LoC is equally applicable for both soft and hard RTSs. However, optimizing RRC with respect to LoC is more important for hard RTSs where obtaining the maximal LoC is one of the major objectives. Using the expression that calculates the LoC, we evaluated the LoC when RRC is optimized for soft RTSs. The conclusion is that the probabilistic guarantees, such as LoC, are very poor when RRC is optimized solely for soft RTSs. This shows an evident gap between the optimization objectives for soft and hard RTSs and motivates the need for other optimization objectives.

For the optimization of *Guaranteed Completion Time*, we analyzed the impact of the number of checkpoints on the guaranteed completion time GCT_δ, *i.e.* an instance of the completion time such that the probability that a job completes within the given GCT_δ satisfies a given LoC requirement δ. With respect to this, we have provided a method that finds the optimal number of checkpoints that results in the minimal GCT_δ. Obtaining the minimal GCT_δ bridges the gap between soft and hard RTSs as it takes into consideration both the completion time and a given LoC requirement.

Part II

Non-Equidistant Checkpointing

In this part, we discuss optimization of RRC while considering that the checkpoints are not evenly distributed, *i.e.* non-equidistant checkpointing. For non-equidistant checkpointing, two different optimization objectives are considered, *i.e.* (1) *Average Execution Time* and (2) *Level of Confidence*. One chapter is dedicated to each of the two optimization objectives. For each optimization objective, the problem formulation is stated and a solution is provided. Each solution is validated with experimental results. Finally, a summary of the part is presented.

7

Average Execution Time

In Chapter 3, we showed that it is possible to find the optimal number of checkpoints that leads to the minimal AET. There, we considered that the checkpoints are evenly distributed throughout the execution of the job. However, in practice, distributing the checkpoints evenly may not be desirable. For example, if errors occur in bursts, it is not desirable to have all the execution segments to be of the same size. Instead, it is more desirable to have less checkpoints, and thus larger execution segments, when errors occur seldomly, and have more checkpoints, *i.e.* shorter execution segments, when errors occur more often. Furthermore, in Chapter 3, we relied on the assumption that the error (or error-free) probability was given. However, not always, the probability of errors is known in advance. Further, the error probability depends on many factors, including influence from the environment where the system operates, which makes it difficult to estimate the error probability during runtime.

In this chapter, we demonstrate that inaccurate estimates of the error probability lead to loss of performance. To avoid inaccurate estimates, we propose two techniques that provide on-line estimation of the error probability. Further, these techniques employ adjustment of the RRC scheme to regain the lost performance, *i.e.* reduce the AET. The proposed techniques are: Periodic Probability Estimation and Aperiodic Probability Estimation. Using a simulator tool that has been developed to enable experimentation, the proposed techniques are evaluated, and the results show that the proposed techniques provide useful estimates of the error probability leading to near-optimal performance of RRC.

The rest of this chapter is organized as follows. The problem discussed in this chapter is presented in Section 7.1. Section 7.2 demonstrates the need of accurate error probability estimates. Two estimation techniques are presented in Section 7.3. Finally, experimental results are presented in Section 7.4.

7.1. PROBLEM FORMULATION

In this chapter, we address the following problem. Given the following inputs:

- a job with a processing time T, and
- an initial error probability estimate, q

adjust the RRC scheme, using an on-line error probability estimation technique, such that the AET is reduced.

7.2. MOTIVATION

In this section, we demonstrate the importance of having accurate error probability estimates by presenting the impact of inaccurate error probability estimates on the number of checkpoints and the resulting AET. As we have already shown in Chapter 3, the optimal number of checkpoints n_c^* depends on the error-free probability, and we show this expression in Eq. (7.1).

$$n_c^* = -(\ln P_T) + \sqrt{(\ln P_T)^2 - \frac{2 \times T \times (\ln P_T)}{\tau_s + \tau_c + \tau_u + 4 \times \tau_b}} \tag{7.1}$$

In Eq. (7.1), P_T denotes the probability that no errors occur (error-free probability) in a processing node within an interval of length T, where T denotes the processing time, *i.e.* the fault-free execution time for a job when RRC is not used. From Eq. (7.1) we observe that the optimal number of checkpoints depends on P_T, T, and the checkpointing overhead $\tau = \tau_s + \tau_c + \tau_u + 4 \times \tau_b$. Since the checkpointing overhead is architecture dependent, *i.e.* it depends on the system architecture, for a given architecture, the checkpointing overhead is constant, and therefore the optimal number of checkpoints is a function that depends solely on the processing time T and the error-free probability P_T. If instead of the error-free probability P_T, the error probability is given, *i.e.* $Q_T = 1 - P_T$, the optimal number of checkpoints n_c^* can be expressed as a function of the error probability Q_T. This is shown in Eq. (7.2).

$$n_c^*(Q_T, T) = -(\ln(1 - Q_T)) + \sqrt{(\ln(1 - Q_T))^2 - \frac{2 \times T \times (\ln(1 - Q_T))}{\tau_s + \tau_c + \tau_u + 4 \times \tau_b}} \tag{7.2}$$

In Chapter 3, we have also defined the expression for calculating the minimal AET which depends on the error-free probability P_T. In a similar way, as for the optimal number of checkpoints, we re-write the expression for the

minimal AET such that it becomes a function of the error probability Q_T. The expression is given in Eq. (7.3).

$$AET(Q_T, T) = \frac{T + n_c^*(Q_T, T) \times (\tau_s + \tau_c + \tau_u + 4 \times \tau_b)}{\sqrt[n_c^*(Q_T,T)]{(1 - Q_T)^2}} \tag{7.3}$$

However, not always, the *real* (actual) error probability is known at design time, and further it can vary over the product's lifetime (time in operation). Because of this fact, it is common that the *initial* chosen error probability, which is used when calculating the optimal number of checkpoints that results in the minimal AET, will differ from the *real* error probability. Therefore, the *initial* chosen error probability value represents an inaccurate error probability estimate.

The *inaccurate* error probability estimate, results in a value for n_c^* which will differ from the optimal, and thus lead to an AET larger than the optimal. In such case, when the estimated error probability q is used to obtain the optimal number of checkpoints n_c^* (Eq. (7.2)), the AET is calculated as shown in Eq. (7.4).

$$AET_{est_q}(Q_T, T, q) = \frac{T + (4 \times \tau_b + \tau_s + \tau_c + \tau_u) \times n_c^*(q, T)}{\sqrt[n_c^*(q,T)]{(1 - Q_T)^2}} \tag{7.4}$$

It should be noted in Eq. (7.4) that the AET is equal to the minimal AET when the estimated error probability q is equal to the *real* error probability Q_T, and thus $AET_{est_q}(Q_T, T, Q_T) = AET(Q_T, T)$.

To quantify the impact of an inaccurate error probability estimate, we use the expression presented in Eq. (7.5):

$$AET_{dev}(Q_T, T, q) = \frac{AET_{est_q}(Q_T, T, q) - AET(Q_T, T)}{AET(Q_T, T)} \times 100\% \tag{7.5}$$

where Q_T is the *real* error probability, and q is the *estimated* error probability. This equation represents the relative deviation in AET compared to the optimum, when an estimate on the error probability is used in order to obtain the number of checkpoints.

To illustrate the impact of inaccurate estimation of the error probability, in Figure 7.1, we show the performance degradation (AET_{dev}) at various estimated error probabilities, for a job with a processing time T=1000 t.u. Three cases are illustrated in Figure 7.1, where $Q_T = 0.5$ in the first case, $Q_T = 0.2$ in the second case, and $Q_T = 0.1$ for the third case. In Figure 7.1, the x-axis represents the estimated error probability q, while the y-axis shows the relative deviation in AET, *i.e.* $AET_{dev}(Q_T, T, q)$ calculated according to Eq. (7.5).

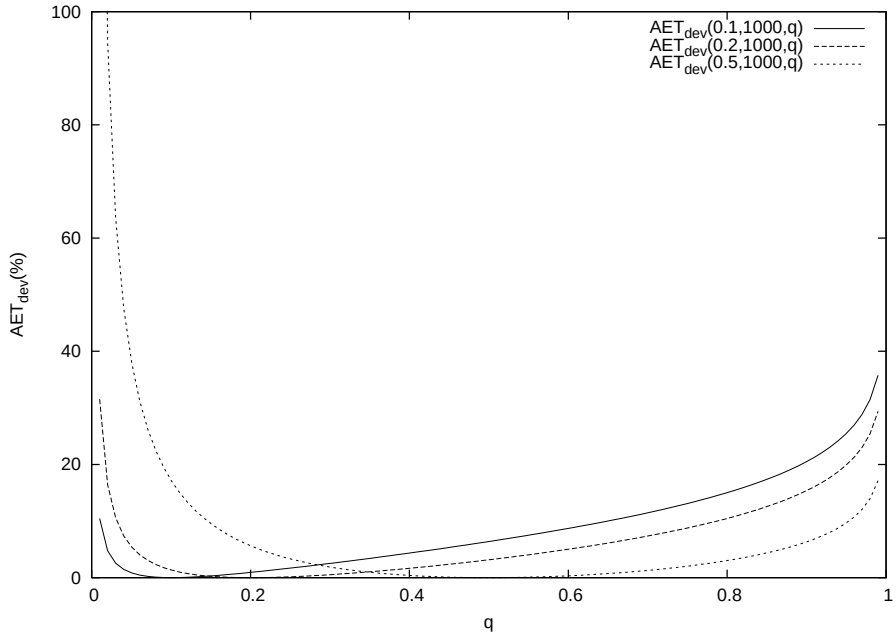


Figure 7.1.: Impact of inaccurate error probability estimation on AET in terms of relative deviation from the optimal AET (%)

As can be seen from Figure 7.1, each curve shows no deviation in AET when the estimated error probability q is equal to the *real* error probability Q_T. However, as soon as $q \neq Q_T$, $AET_{dev}(Q_T, T, q)$ is increased. This means that assuming an error probability other than the real one leads to an AET which is not the optimal. The increase in AET due to inaccurate error probability estimation represents the loss of performance.

7.3. ERROR PROBABILITY ESTIMATION AND CORRESPONDING ADJUSTMENT

In this section, we present approaches that estimate the error probability with the aim to adjust and optimize RRC during operation. To make use of the estimates on the error probability, we need to estimate the error probability during operation. One way to achieve this is to extend the architecture described earlier in Chapter 2 (see Figure 2.1) by employing a history unit that keeps track on the number of successful execution segments n_s, and the number of erroneous execution segments n_e. By using these statistics, the error probability can be estimated during time either periodically or aperiodically.

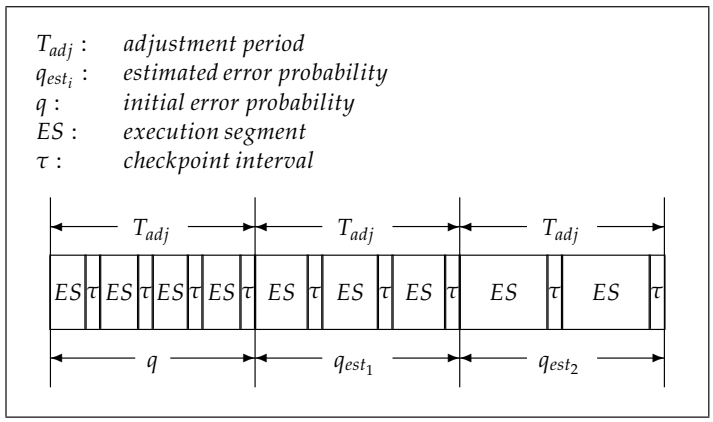


Figure 7.2.: Graphical presentation of PPE

Thus, we come up with one periodic approach, which we address as Periodic Probability Estimation (PPE), and one aperiodic, which we address as Aperiodic Probability Estimation (APE). For both approaches, we need some initial parameters, *i.e.* initial estimate on error probability and an adjustment period. It should be noted, that the adjustment period is kept constant for PPE, while for APE it is tuned over time.

PERIODIC PROBABILITY ESTIMATION PPE assumes a fixed T_{adj} and elaborates on q_{est} as:

$$q_{est} = \frac{n_e}{n_e + n_s} \tag{7.6}$$

where n_s is the number of successful execution segments and n_e is the number of erroneous execution segments. As can be seen from Figure 7.2, estimates on the error probability q_{est} are calculated periodically at every T_{adj}. The value of the most recent estimate q_{est} is used to calculate the optimal number of checkpoints $n_c^\dagger = n_c^*(q_{est}, T_{adj})$. During the next adjustment period T_{adj}, $n_c^\dagger$ equidistant checkpoints are taken. Therefore, the checkpoint frequency, *i.e.* number of checkpoints during a time interval, changes according to the changes of the error probability estimates.

APERIODIC PROBABILITY ESTIMATION APE elaborates on both T_{adj} and q_{est}. The idea behind this approach comes from the following discussion. As this approach is an estimation technique, it is expected that during operation the estimates will converge to the real values and therefore, we should expect changes on the estimated error probability during time. These changes can guide how to change the checkpointing scheme. If the estimates

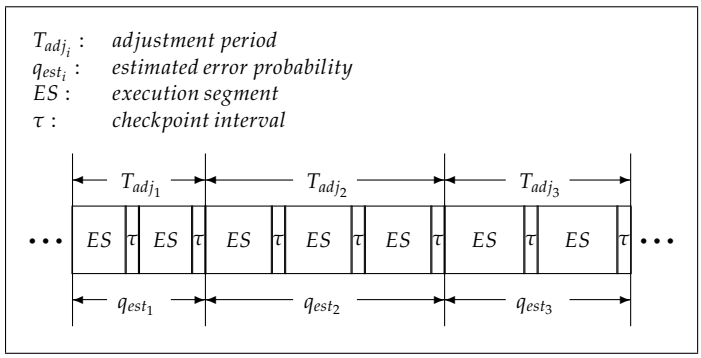


Figure 7.3.: Graphical presentation of APE

on error probability start decreasing, this implies that less errors are occurring and then, less frequent checkpointing is required. Thus, in such case, the adjustment period is increased. On the other hand, if the estimates on error probability start increasing, this implies that errors occur more frequently and therefore, to reduce the time spent in re-execution more frequent checkpointing is required. Thus, in such case, the adjustment period is decreased.

If the estimates on the error probability have not changed during two successive adjustment periods, it means that during both adjustment periods the system has employed a number of checkpoints larger than the optimal one. This can be observed by the following inequality:

$$2 \times n_c^*(Q_T, T_{adj}) > n_c^*(Q_T, 2 \times T_{adj}) \tag{7.7}$$

In APE, the error probability is estimated in the same manner as PPE, *i.e.* by using Eq. (7.6). What distinguishes APE from PPE, is that the adjustment period T_{adj} is updated during time by using an update factor α. The update factor α is used to either increase or decrease the adjustment period. The scheme that is used for updating the adjustment period is described in Eq. (7.8).

$$\begin{aligned} &if \qquad q_{est_{i+1}} > q_{est_i} \qquad then \\ &\quad T_{adj_{i+1}} = T_{adj_i} - T_{adj_i} \times \alpha \\ &else \\ &\quad T_{adj_{i+1}} = T_{adj_i} + T_{adj_i} \times \alpha \end{aligned} \tag{7.8}$$

The APE approach is illustrated in Figure 7.3. After every T_{adj} time units, a new error probability estimate $q_{est_{i+1}}$ is computed by using the Eq. (7.6). The latest estimate $q_{est_{i+1}}$ is then compared against the most recent value q_{est_i}. If the estimates on the error probability increase, meaning that during the last adjustment period T_{adj_i} more errors have occurred, the next adjustment

period $T_{adj_{i+1}}$ is decreased to avoid expensive re-executions. However, if the estimates on error probability decrease or remain the same, meaning that less or no errors have occurred during the last adjustment period T_{adj_i}, the next adjustment period $T_{adj_{i+1}}$ is increased to avoid excessive checkpointing.

7.4. EXPERIMENTAL RESULTS

The purpose of the results presented in this section is to show that both approaches, *i.e.* Periodic Probability Estimation (PPE) and Aperiodic Probability Estimation (APE), make a good estimation on the error probability, and by doing so, they significantly reduce the AET compared against a Baseline Approach (BA) which takes checkpoints at a constant frequency, and does not perform any adjustments during the execution of a job. For that reason, we conducted the following two experiments:

- P1: evaluate and compare the AET for PPE, APE and BA when the *real* error probability is constant;
- P2: evaluate and compare the AET for PPE, APE and BA when the *real* error probability changes over time.

To conduct these experiments, we have developed a simulator that simulates the execution of a job when one of the three approaches (PPE, APE or BA) is used. There are two types of inputs that are given to the simulator, *designer* inputs and *environmental* inputs. The *designer* inputs refer to inputs that initialize the system, *i.e.* the initial estimate on the error probability q, the adjustment period T_{adj}, and the update factor α. The *environmental* inputs refer to the *real* error probability Q_T and the processing time T. Important feature of the simulator is that the *real* error probability Q_T can be modeled as a function that changes over time. The reason for this is that the error probability does not need to be constant, and it may change over time. The input Q_T is used to inject errors while simulating the approaches. The output of the simulator is the execution time, *i.e.* the time required for a job with a processing time T to complete given that errors occur according to Q_T, when one of the three approaches (PPE, APE or BA) is used. Next, we discuss how each of the three approaches uses the inputs that are given to the simulator.

The BA takes the following *designer* inputs, namely the initial estimate on the error probability q, and the adjustment period T_{adj}, and it computes the number of checkpoints $n_c^\dagger = n_c^*(q, T_{adj})$ for these inputs by using Eq. (7.2). Further, it takes checkpoints at a constant frequency $n_c^\dagger / T_{adj}$, and no adjustments are done during the execution of the job. The PPE approach takes the following *designer* inputs, namely the initial estimate on the error probability q, and the adjustment period T_{adj}. In contrast to BA, PPE estimates the error

probability over time, and updates the number of checkpoints after each adjustment period T_{adj} (observe Figure 7.2). The APE approach takes all *designer* inputs, namely the initial estimate on the error probability q, the adjustment period T_{adj}, and the update factor α, and it updates the checkpointing scheme as shown in Figure 7.3.

Since for the experiments it is required to obtain the AET for each of the approaches, the AET is obtained by averaging the outputs of the simulator after repeating the simulation for the same inputs 1000 times. For example, to obtain the AET when BA is used, for a given set of *designer* and *environmental* inputs, we simulate the execution of the job according to BA 1000 times while using the same set of inputs in each simulation, and we obtain the AET by calculating the average of the outputs obtained from each simulation. Next, we provide the results for the two experiments P1 and P2.

For P1, we compare the AET obtained from the three simulated approaches: PPE, APE, and BA against the optimal AET for different initial estimates on the error probability q. The optimal AET is obtained by evaluating Eq. (7.3) for the given *environmental* inputs, *i.e.* Q_T and T. We made several experiments by varying both the *designer* and *environmental* inputs. In Figure 7.4, we present the relative deviation from the optimal AET for the three approaches: PPE, APE, and BA, for the following set of inputs:

- a processing time $T = 1000000$ time units (t.u.),
- a *real* error probability $Q_T = 0.01$,
- an adjustment period $T_{adj} = 1000$ t.u., and
- an update factor $\alpha = 0.15$.

The horizontal axis, in Figure 7.4, represents the difference between the initial estimate on the error probability q and the *real* error probability Q_T, while the vertical axis represents the relative deviation from the optimal AET, expressed in percent. One can observe from Figure 7.4 that APE and PPE do not depend significantly on the initial estimate on the error probability q. Both APE and PPE always perform better than the BA approach. The small relative deviation from the optimal AET observed for PPE and APE shows that both approaches make a good estimation on the *real* error probability. Further, Figure 7.4 shows that APE performs slightly better than PPE.

For P2, we evaluate the approaches in terms of relative deviation with respect to the processing time T, when the *real* error probability is not constant. For this purpose, we defined three error probability profiles according to which the error probability changes over time, and then we ran simulations for each of these profiles. The error probability profiles: $Q_1(t)$, $Q_2(t)$, and $Q_3(t)$ are presented in Table 7.1. Important to note is that the error prob-

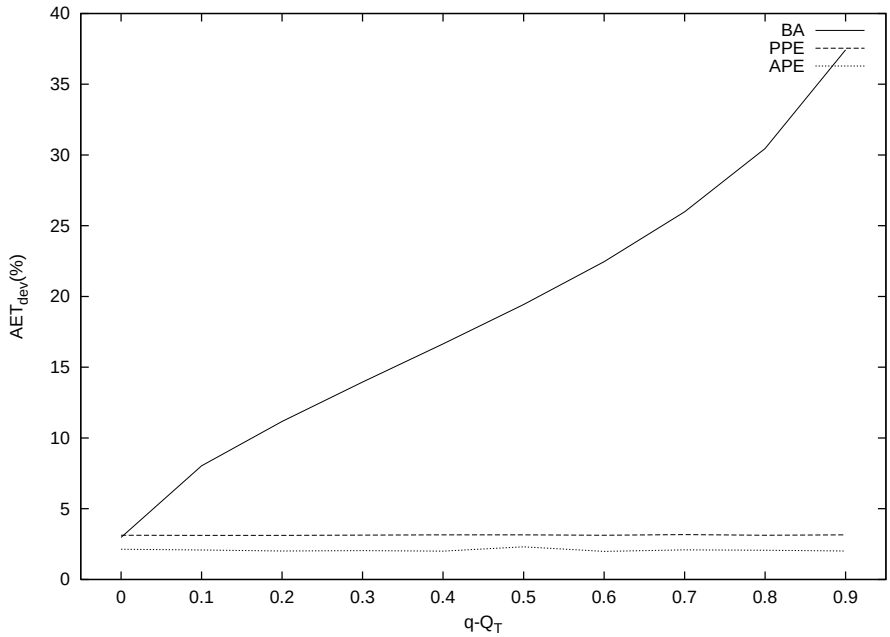


Figure 7.4.: Relative deviation from the optimal AET (%) for a constant *real* error probability $Q_T = 0.01$

$$Q_1(t) = \begin{cases} 0.01, & 0 \leq t \mod T < 200000 \\ 0.02, & 200000 \leq t \mod T < 400000 \\ 0.03, & 400000 \leq t \mod T < 600000 \\ 0.02, & 600000 \leq t \mod T < 800000 \\ 0.01, & 800000 \leq t \mod T < 1000000 \end{cases}$$

$$Q_2(t) = \begin{cases} 0.02, & 0 \leq t \mod T < 350000 \\ 0.01, & 350000 \leq t \mod T < 650000 \\ 0.02, & 650000 \leq t \mod T < 1000000 \end{cases}$$

$$Q_3(t) = \begin{cases} 0.01, & 0 \leq t \mod T < 90000 \\ 0.10, & 90000 \leq t \mod T < 100000 \end{cases}$$

Table 7.1.: Three error probability profiles: $Q_1(t)$, $Q_2(t)$, and $Q_3(t)$

ability profiles are repeated periodically over time with a period equal to T. According to the error probability profile $Q_1(t)$ the error probability increases from 0.01 to 0.03 in the first half of the period, and then it decreases from 0.03 to 0.01 in the second half of the period (see Figure 7.5(a)). According to the error probability profile $Q_2(t)$, the error probability drops from 0.02 to 0.01 during the second third of the period, and rises back to 0.02 in the last third of the period (see Figure 7.5(b)). The error probability profile $Q_3(t)$ models the scenario where errors occur in burst, *i.e.* the error probability is much higher (0.1) during a short interval (the first 90000 t.u.) at the beginning of the period (the period is 1000000 t.u.), but it drops to a much lower value (0.01) in the rest of the period (see Figure 7.5(c)). For a given error probability profile $Q_i(t)$ and the following set of inputs:

- a processing time $T = 1000000$ t.u.,
- an initial estimate on the error probability $q = Q_i(0)$,
- an adjustment period $T_{adj} = 1000$ t.u., and
- an update factor $\alpha = 0.15$,

in Table 7.2, we present the relative deviation of the obtained AET, for each of the simulated approaches, with respect to the processing time T. The results are expressed as a percentage. As can be seen from Table 7.2, both PPE and APE perform far better than BA, with a very small deviation with respect to the processing time. Again we notice that APE gives slightly better results than PPE approach. From the experimental results presented in this section, we conclude that both approaches PPE and APE make a good estimation on the *real* error probability and as a consequence, the obtained AET is very close to (1) the optimal AET, when the *real* error probability is constant or (2) the processing time T, when the *real* error probability changes over time according to a given error probability profile.

Probability Profile $Q_i(t)$	Approaches		
	Baseline	PPE	APE
$Q_1(t)$	55.93%	4.50%	2.84%
$Q_2(t)$	50.69%	4.53%	2.74%
$Q_3(t)$	56.02%	4.65%	2.50%

Table 7.2.: Relative deviation from the processing time T (%) for the three error probability profiles $Q_1(t)$, $Q_2(t)$, and $Q_3(t)$

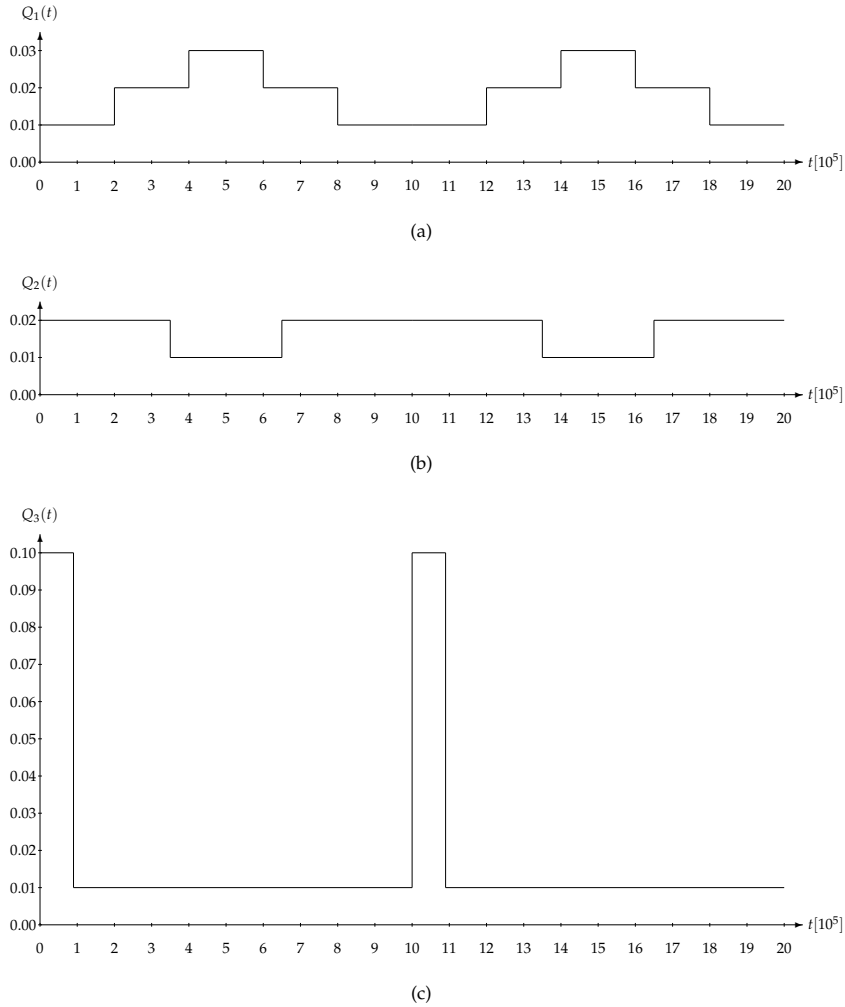


Figure 7.5.: Illustration of the three error probability profiles (a) $Q_1(t)$, (b) $Q_2(t)$, and (c) $Q_3(t)$

8

Level of Confidence

In Chapter 4, we showed how to find the optimal number of checkpoints that results in the maximal LoC for a single job while assuming that the checkpoints are evenly distributed. The assumption of equidistant checkpointing dictates that the execution segments must be of the same size and thus, a checkpoint has to be taken at an exact point in time during the execution of a job. However, in practice, the execution of a job consists of executing a set of instructions where an on-going instruction cannot be interrupted and thus, a checkpoint can only be taken once an on-going instruction has completed. This conflicts with the assumption of equidistant checkpointing where a checkpoint has to be taken at an exact point in time, and motivates the need to study what is the impact on the LoC when the checkpoints are not evenly distributed, *i.e.* non-equidistant checkpointing.

In this chapter, we discuss optimization of RRC with respect to LoC when considering non-equidistant checkpointing. The chapter is organized as follows. In Section 8.1, we present the problem formulation. The motivation is provided in Section 8.2. An expression to evaluate the LoC when the checkpoints are not evenly distributed is derived and presented in Section 8.3. To obtain the optimal solution for the stated problem, in Section 8.4, we discuss an exhaustive search method and show that the complexity of this method is extremely high. Therefore, in Section 8.5, we present the heuristic method Clustered Checkpointing. Finally, experimental results are presented in Section 8.6.

8.1. PROBLEM FORMULATION

In this section, we address the following problem. Given the following inputs:

- a processing time T,
- a deadline D,
- a checkpointing overhead τ,
- a probability P_T that no errors occur within an interval of length T, and
- a number of checkpoints n_c,

find the optimal distribution of the given number of checkpoints n_c that results in the maximal LoC with respect to the given deadline D, while assuming that a checkpoint can only be taken at an integer time unit.

8.2. MOTIVATION

In equidistant checkpointing, the checkpoints are evenly distributed. Therefore, applying equidistant checkpointing with n_c checkpoints for a job with a processing time T results in n_c execution segments with size $\frac{T}{n_c}$. While equidistant checkpointing is justified from a theoretical point of view, in practice, this may not always be possible. Since the execution of a job consists of executing a set of instructions where an on-going instruction cannot be interrupted, a checkpoint can only be taken once an on-going instruction has completed. Therefore, if each instruction takes one time unit, a checkpoint can be taken at any integer time unit (integer sized execution segments). The drawback with this is that it is not possible to distribute the checkpoints evenly, if the processing time, measured in time units, is not a multiple of the number of checkpoints. For example, if $T = 10$ t.u. (time units) and $n_c = 3$, then the limitation of integer sized execution segments results in having execution segments of different size, *i.e.* two execution segments of size 3 t.u. and one execution segment of size 4 t.u. Hence, in practice, equidistant checkpointing may result in execution segments of different size (4 t.u and 3 t.u. for the example above). This motivates the need for a mathematical framework for evaluation of LoC when the checkpoints are not evenly distributed. Having such a framework makes it possible to answer the question: can a non-equidistant distribution result in a higher LoC than equidistant checkpointing?

8.3. EVALUATION OF LEVEL OF CONFIDENCE

In this section, we show how to calculate the LoC with respect to a given deadline, for a given distribution of n_c checkpoints. The distribution of the n_c checkpoints is represented by a vector $\bar{n}_c$ of size n_c, where the i-th element (ES_i) in the vector represents the size of the i-th execution segment, *i.e.* $\bar{n}_c = [ES_1, ES_2..., ES_{n_c}]$. Similarly to equidistant checkpointing, the LoC is calculated as a sum of probabilities that a job completes at time points lower or equal to the given deadline. While for equidistant checkpointing, for a given number of checkpoints, the job is expected to complete only at discrete equidistant time moments, in the case of non-equidistant checkpointing, the job is expected to complete at discrete time moments which are no longer equidistant. This follows from the fact that in non-equidistant checkpointing, the execution segments are not of the same size. Thus, re-execution of different execution segments has different impact on the completion time due to the different re-execution cost. The re-execution cost of an execution segment is calculated as the sum of the size of the execution segment ES_i, and the checkpointing overhead τ. From the discussion above, it follows that the completion time, for a given distribution $\bar{n}_c$ of n_c checkpoints, can be expressed as a discrete variable ${}^{\bar{n}_c}t_{\bar{k}}$, where $\bar{k}$ denotes a re-execution vector. The re-execution vector $\bar{k}$ represents a vector of size n_c and each element k_i in $\bar{k}$ indicates the number of re-executions of the i-th execution segment. The expression in Eq. (8.1) is used to calculate ${}^{\bar{n}_c}t_{\bar{k}}$.

$$ {}^{\bar{n}_c}t_{\bar{k}} = T + n_c \times \tau + \sum_{i=1}^{n_c} (ES_i + \tau) \times k_i \tag{8.1} $$

The first two terms in Eq. (8.1), *i.e.* $T + n_c \times \tau$, express the time required for the job to complete when no errors occur in any of the n_c execution segments, *i.e.* $k_i = 0, \ \forall \ i \ \in [1, n_c]$. The third term in Eq. (8.1) expresses the cost of re-execution due to errors in any of the n_c execution segments and it is calculated as a sum of the individual re-execution costs of each of the n_c execution segments. The individual re-execution cost of an execution segment ES_i is calculated as a product of the re-execution cost, *i.e.* $ES_i + \tau$, and the number of re-executions k_i of the particular execution segment. For each discrete value of ${}^{\bar{n}_c}t_{\bar{k}}$ there exists a probability that a job completes at that given instance in time. We denote the probability that a job completes at ${}^{\bar{n}_c}t_{\bar{k}}$ with $p_{\bar{n}_c}(\bar{k})$. Next, we elaborate how to calculate $p_{\bar{n}_c}(\bar{k})$.

When a job completes at time ${}^{\bar{n}_c}t_{\bar{k}}$, it means that each execution segment i has been executed exactly $k_i + 1$ times, and only the last execution has been successful. Thus, for the i-th execution segment there have been k_i erroneous executions and a single successful execution. Since we assume that (1) the occurrence of errors is an independent event and (2) the probability P_T that

no errors occur within an interval of length T is given, we can calculate the probability of a successful execution segment of size ES_i. Observe that an execution segment can only be successfully executed if no errors have occurred in any of the processing nodes during the execution of the segment. Since P_T represents the probability that no errors occur in a processing node within an interval of length T, the probability that no errors occur in one processing node during the execution of an execution segment of size ES_i is evaluated as $P_T{}^{\frac{ES_i}{T}}$. We denote with P_{ϵ_i} the probability of a successful execution segment of size ES_i and we calculate it with the following expression:

$$P_{\epsilon_i} = P_T{}^{\frac{ES_i}{T}} \times P_T{}^{\frac{ES_i}{T}} = P_T{}^{\frac{2\times ES_i}{T}} \tag{8.2}$$

Since in Eq. (8.2) we have the expression to calculate the probability of a successful execution segment of size ES_i, the probability of an erroneous execution segment of size ES_i can be calculated as $1 - P_{\epsilon_i}$. Given the probability of a successful and erroneous execution segment, *i.e.* P_{ϵ_i} and $1 - P_{\epsilon_i}$, the probability that a job completes at ${}^{\bar{n}_c}t_{\bar{k}}$ is calculated with the following expression:

$$p_{\bar{n}_c}(\bar{k}) = \prod_{i=1}^{n_c} P_{\epsilon_i} \left(1 - P_{\epsilon_i}\right)^{k_i} \tag{8.3}$$

The expression in Eq. (8.3) can be reduced due to the fact that the product of all P_{ϵ_i} terms can be replaced by a single constant. This is shown in the following equation:

$$\begin{aligned}
p_{\bar{n}_c}(\bar{k}) &= \prod_{i=1}^{n_c} P_{\epsilon_i} \left(1 - P_{\epsilon_i}\right)^{k_i} \\
&= \prod_{i=1}^{n_c} P_{\epsilon_i} \prod_{i=1}^{n_c} \left(1 - P_{\epsilon_i}\right)^{k_i} \\
&= \prod_{i=1}^{n_c} P_T{}^{\frac{2\times ES_i}{T}} \prod_{i=1}^{n_c} \left(1 - P_{\epsilon_i}\right)^{k_i} \\
&= P_T{}^{\frac{\sum_{i=1}^{n_c} 2\times ES_i}{T}} \prod_{i=1}^{n_c} \left(1 - P_{\epsilon_i}\right)^{k_i} \\
&= P_T{}^{\frac{2\times\sum_{i=1}^{n_c} ES_i}{T}} \prod_{i=1}^{n_c} \left(1 - P_{\epsilon_i}\right)^{k_i}
\end{aligned} \tag{8.4}$$

Since the sum of all execution segments ES_i is equal to the processing time T,

we replace the sum in Eq. (8.4) with T. Thus, we get the following expression:

$$p_{\bar{n}_c}(\bar{k}) = {P_T}^{\frac{2\times T}{T}} \prod_{i=1}^{n_c} (1 - P_{\epsilon_i})^{k_i}$$
$$= {P_T}^2 \prod_{i=1}^{n_c} (1 - P_{\epsilon_i})^{k_i} \tag{8.5}$$

From Eq. (8.5) it follows that the probability that a job completes without any re-executions (no errors), *i.e.* $k_i = 0,\ \forall\ i\ \in [1, n_c]$, is constant and it is the same as the probability that a job completes at time ${}^{n_c}t_0$ when equidistant checkpointing is used, *i.e.* $p_{n_c}(0) = {P_T}^2$ (see Eq. (4.6) in Section 4.1.2 in Chapter 4). However, if errors occur and therefore, some execution segments are re-executed, the probability depends on the number of re-executions and on the size of the execution segments which have been re-executed.

Eq. (8.5) represents the probability distribution function $p_{\bar{n}_c}(\bar{k})$ and it computes the probability that a job completes at a given time instance ${}^{\bar{n}_c}t_{\bar{k}}$. To compute the LoC with respect to a given deadline it is required to sum the probabilities of all ${}^{\bar{n}_c}t_{\bar{k}}$ which are lower or equal to the deadline. We denote with $\Lambda_{\bar{n}_c}(D)$ the LoC with respect to a given deadline D, for a given distribution $\bar{n}_c$ of n_c checkpoints, and it is calculated with the following expression:

$$\Lambda_{\bar{n}_c}(D) = \sum_{\bar{k}}^{{}^{\bar{n}_c}t_{\bar{k}} \leq D} p_{\bar{n}_c}(\bar{k}) \tag{8.6}$$

From Eq. (8.6), one observes that to calculate the LoC it is required to obtain all the discrete values of the completion time ${}^{\bar{n}_c}t_{\bar{k}}$ that are lower or equal to the given deadline. For a given distribution $\bar{n}_c$, the completion time ${}^{\bar{n}_c}t_{\bar{k}}$ varies with different re-execution vectors $\bar{k}$ and only a set of these vectors produce discrete values of the completion time which are lower or equal to the given deadline. We refer to these re-execution vectors as *valid* re-execution vectors. To obtain the set of *valid* re-execution vectors we use the function $f(\bar{k}, i)$ given in Eq. (8.8). For a given *valid* re-execution vector, $f(\bar{k}, i)$ produces a new *valid* re-execution vector if such vector exists, or it returns an empty set otherwise. The function requires two input parameters: a *valid* re-execution vector ($\bar{k}$) and an *update* index (i). For the given parameters, $f(\bar{k}, i)$ first computes a tentative re-execution vector $\bar{k}^N$ by incrementing k_i by one and setting to zero all k_j where $j > i$ (see Eq. (8.7)). If $\bar{k}^N$ is a *valid* re-execution vector and $i > 0$, then $f(\bar{k}, i)$ returns $\bar{k}^N$. Otherwise, $f(\bar{k}, i)$ proceeds with a recursive call $f(\bar{k}, i-1)$. If the function is invoked with $i = 0$, $f(\bar{k}, i)$ returns an empty set, meaning that all *valid* re-execution vectors have been identified. To identify the entire set of *valid* re-execution vectors, $f(\bar{k}, i)$ is iteratively called with these arguments: (1) the most recently obtained *valid* re-execution vector (the initial

valid re-execution vector is $\bar{k} = [0, 0...0]$) and (2) the number of checkpoints n_c is passed as the *update* index.

$$\bar{k}^N = [k1, k2, ...k_{i-1}, k_i + 1, 0...0] \tag{8.7}$$

$$f(\bar{k}, i) = \begin{cases} \bar{k}^N, \text{ iff } \bar{k}^N \text{ is } \textit{valid} \text{ and } i \geq 1 \\ f(\bar{k}, i-1), \text{ iff } \bar{k}^N \text{ is not } \textit{valid} \text{ and } i \geq 1 \\ \emptyset, \text{ iff } i = 0 \end{cases} \tag{8.8}$$

Next, we demonstrate the usage of this function for the following scenario. Given a job with a processing time $T = 600$ t.u., a deadline $D = 1000$ t.u., and a checkpointing overhead $\tau = 20$ t.u., the function $f(\bar{k}, i)$ will be used to obtain all *valid* re-execution vectors for the following distribution of three checkpoints $\bar{n}_c = [150, 200, 250]$. As we mentioned earlier, the function is iteratively invoked with the most recently obtained *valid* re-execution vector (starting with the initial vector $\bar{k} = [0, 0...0]$) and the number of checkpoints. Thus, the function for the given scenario is first invoked with the following inputs $\bar{k} = [0, 0, 0]$ and $i = 3$ (observe that from the given distribution $\bar{n}_c = [150, 200, 250]$ there are three checkpoints). For these inputs, the function first calculates a tentative re-execution vector $\bar{k}^N = [0, 0, 1]$. Using Eq. (8.1), the re-execution vector $\bar{k}^N$ results in a completion time equal to $600 + 3 \times 20 + (250 + 20) = 930$ t.u. which is lower than the given deadline D. Therefore, the tentative re-execution vector $\bar{k}^N = [0, 0, 1]$ is a *valid* re-execution vector, and it represents the output of $f(\bar{k}, i)$ which is invoked with $\bar{k} = [0, 0, 0]$ and $i = 3$. Next, $f(\bar{k}, i)$ is invoked with the most recently obtained *valid* re-execution vector, *i.e.* $\bar{k} = [0, 0, 1]$ and $i = 3$. For these arguments, the function first calculates $\bar{k}^N = [0, 0, 2]$. However, $\bar{k}^N$ is not a *valid* re-execution vector due to that it results in a completion time $600 + 3 \times 20 + 2 \times (250 + 20) = 1200$ t.u. which violates the given deadline. For that reason, the function steps in a recursive call with the following inputs $\bar{k} = [0, 0, 1]$ and $i = 2$. For these arguments, the function computes $\bar{k}^N = [0, 1, 0]$. Since the completion time for $\bar{k}^N = [0, 1, 0]$ is $600 + 3 \times 20 + (200 + 20) = 880$ t.u. which is lower than the deadline, the function returns the vector $[0, 1, 0]$ as the most recently obtained *valid* re-execution vector. Next, $f(\bar{k}, i)$ is invoked with $\bar{k} = [0, 1, 0]$ and $i = 3$. For these arguments, the function computes $\bar{k}^N = [0, 1, 1]$. However, $\bar{k}^N$ is not a *valid* re-execution vector as it results in a completion time $600 + 3 \times 20 + (200 + 20) + (250 + 20) = 1150$ t.u.. Therefore, the function steps in a recursive call with the following arguments $\bar{k} = [0, 1, 0]$ and $i = 2$, and it computes a $\bar{k}^N = [0, 2, 0]$. The recently computed $\bar{k}^N$ is again not *valid* as it results in a completion time $600 + 3 \times 20 + 2 \times (200 + 20) = 1100$ t.u..

Thus, the function steps into yet another recursive call, this time with the following arguments $\bar{k} = [0, 1, 0]$ and $i = 1$. For these arguments, the function computes $\bar{k}^N = [1, 0, 0]$ which represents a *valid* re-execution vector as the completion time $600 + 3 \times 20 + (150 + 20) = 830$ t.u. does not violate the deadline. Therefore, the function returns the vector $[1, 0, 0]$ as the latest *valid* re-execution vector. Next, the function is invoked with these arguments $\bar{k} = [1, 0, 0]$ and $i = 3$. For this set of arguments, the function computes $\bar{k}^N = [1, 0, 1]$ which is not a *valid* re-execution vector, and therefore forces the function to proceed with a recursive call with these inputs $\bar{k} = [1, 0, 0]$ and $i = 2$. For these arguments, $f(\bar{k}, i)$ computes $\bar{k}^N = [1, 1, 0]$ which again is not a *valid* re-execution vector, forcing the function to step in yet another recursive call, this time with $\bar{k} = [1, 0, 0]$ and $i = 1$. The function computes $\bar{k}^N = [2, 0, 0]$ which is *valid* and it represents the latest *valid* re-execution vector. Next, the function is invoked with these arguments $\bar{k} = [2, 0, 0]$ and $i = 3$. For these arguments, the functions steps into three consecutive recursive calls computing the following not *valid* tentative re-execution vectors $\bar{k}^N = [2, 0, 1]$, $\bar{k}^N = [2, 1, 0]$ and $\bar{k}^N = [3, 0, 0]$. For the latest obtained $\bar{k}^N = [3, 0, 0]$, which happens for input arguments $\bar{k} = [2, 0, 0]$ and $i = 1$, due to the fact that it is not a *valid* re-execution vector, the function is forced into yet another recursive call with these arguments $\bar{k} = [2, 0, 0]$ and $i = 0$. Since the last recursive call invokes the function with an *update* index zero, the function returns an empty set, meaning that $\bar{k} = [2, 0, 0]$ is the last *valid* re-execution vector.

As shown earlier, the function $f(\bar{k}, i)$ finds all *valid* re-execution vectors $\bar{k}$ that are needed to calculate the LoC with respect to the given deadline. The LoC is calculated by performing the following steps. First, the re-execution vector $\bar{k}$ is initialized to zero, *i.e.* $k_i = 0 \; \forall \; i \; \in [1, n_c]$, and then, the following steps are repeated until $\bar{k} \neq \emptyset$: (1) $p_{\bar{n}_c}(\bar{k})$ is calculated according to Eq. (8.5) and added to the sum which calculates the LoC, and (2) $\bar{k} = f(\bar{k}, n_c)$. However, there are two important considerations when evaluating the LoC for a given distribution $\bar{n}_c$ of a number of checkpoints. The first consideration is about distributions which are permutation of each other, and the second consideration is about distributions where a number of execution segments have the same size.

Evaluation of the LoC for distributions which are permutation of each other is redundant. This is due to the fact that the LoC for such distributions is always the same. Therefore, if the LoC is evaluated for a given distribution $\bar{n}_c$, then there is no need to evaluate the LoC for all other distributions which are permutation of $\bar{n}_c$ since all such distributions will results in the same $\Lambda_{\bar{n}_c}(D)$. The reason for this follows from these two important facts (1) the number of terms that are included in the sum that calculates the LoC (the number of *valid* re-execution vectors) is the same for any permutation of a given distribution $\bar{n}_c$, and (2) all the terms that are included in the sum that calculates the LoC

are the same for any permutation of a given distribution $\bar{n}_c$. These facts come from the commutative property of addition and multiplication.

To show that the number of *valid* re-execution vectors (the number of terms included in the sum that calculates the LoC) for a given distribution $\bar{n}_c$ is the same as the number of *valid* re-execution vectors for any permutation of $\bar{n}_c$, we make use of the commutative property of addition. Consider a given distribution $\bar{n}_c$ and a *valid* re-execution vector $\bar{k}$. For the given $\bar{n}_c$ and $\bar{k}$, ${}^{\bar{n}_c}t_{\bar{k}}$ can be computed by using Eq. (8.1), and thus we get:

$${}^{\bar{n}_c}t_{\bar{k}} = T + n_c \times \tau + k_1(ES_1 + \tau) + k_2(ES_2 + \tau) + ... + k_{n_c}(ES_{n_c} + \tau) \quad (8.9)$$

Due to the commutative property of addition, the right-hand side of Eq. (8.9) can be re-written with the following expression:

$$\begin{aligned} & T + n_c \times \tau + k_{n_c}(ES_{n_c} + \tau) + ... + k_2(ES_2 + \tau) + k_1(ES_1 + \tau) \\ = \quad & T + n_c \times \tau + \sum_{i=1}^{n_c} k_i^{\dagger}(ES_i^{\dagger} + \tau) \end{aligned} \quad (8.10)$$

where $ES_i^{\dagger} = ES_{n_c-i+1}$ and $k_i^{\dagger} = k_{n_c-i+1}$. Using the new notations $ES_i^{\dagger}$ and $k_i^{\dagger}$, the right-hand side of Eq. (8.9) is equal to ${}^{\bar{n}_c^{\dagger}}t_{\bar{k}^{\dagger}}$. Hence, ${}^{\bar{n}_c}t_{\bar{k}} = {}^{\bar{n}_c^{\dagger}}t_{\bar{k}^{\dagger}}$. This shows that if $\bar{k}$ is a *valid* re-execution vector for a distribution $\bar{n}_c$, then for each distribution $\bar{n}_c^{\dagger}$ which is a permutation of $\bar{n}_c$ there exists a *valid* re-execution vector $\bar{k}^{\dagger}$ where $\bar{k}^{\dagger}$ is obtained from $\bar{k}$ by applying the same permutation that is used to construct $\bar{n}_c^{\dagger}$. By this, we conclude that the number of *valid* re-execution vectors (the number of terms included in the sum that calculates the LoC) for a given distribution $\bar{n}_c$ is the same as the number of *valid* re-execution vectors for any permutation of $\bar{n}_c$.

To show that all the terms that are included in the sum that calculates the LoC are the same for any permutation of a given $\bar{n}_c$ we make use of the commutative property of multiplication. Consider a given distribution $\bar{n}_c$ and a *valid* re-execution vector $\bar{k}$. To calculate $p_{\bar{n}_c}(\bar{k})$ we use the Eq. (8.5), and thus we get:

$$p_{\bar{n}_c}(\bar{k}) = {P_T}^2 \left(1 - P_{\epsilon_1}\right)^{k_1} \left(1 - P_{\epsilon_2}\right)^{k_2} ... \left(1 - P_{\epsilon_{n_c}}\right)^{k_{n_c}} \quad (8.11)$$

Due to the commutative property of multiplication, the right-hand side of Eq. (8.11) can be re-written as:

$$\begin{aligned} & {P_T}^2 \left(1 - P_{\epsilon_{n_c}}\right)^{k_{n_c}} ... \left(1 - P_{\epsilon_2}\right)^{k_2} \left(1 - P_{\epsilon_1}\right)^{k_1} \\ & \qquad = {P_T}^2 \prod_{i=1}^{n_c} \left(1 - P_{\epsilon_i}^{\dagger}\right)^{k_i^{\dagger}} \end{aligned} \quad (8.12)$$

where $P_{\epsilon_i}^{\dagger} = P_{\epsilon_{n_c-i+1}}$ and $k_i^{\dagger} = k_{n_c-i+1}$. Using the recent notations, Eq. (8.12) represents $p_{\bar{n}_c^{\dagger}}(\bar{k}^{\dagger})$, and since Eq. (8.12) is equal to Eq. (8.11), it implies that

$p_{\bar{n}_c}(\bar{k}) = p_{\bar{n}_c^\dagger}(\bar{k}^\dagger)$. This shows that all the terms that are included in the sum that calculates the LoC are the same for any permutation of a given distribution $\bar{n}_c$.

For the second consideration, *i.e.* distributions where a number of execution segments have the same size, many different re-execution vectors $\bar{k}$ may provide the same completion time, and for all these re-execution vectors $p_{\bar{n}_c}(\bar{k})$ will be the same. For a given a distribution $\bar{n}_c$, where the first m execution segments have the same size, the re-execution vectors $\bar{k}$ for which the sum of the first m elements k_i is the same will produce the same completion time (see Eq. (8.1)), and furthermore $p_{\bar{n}_c}(\bar{k})$ for all those re-execution vectors $\bar{k}$ will be the same. Since the LoC is computed as a sum of $p_{\bar{n}_c}(\bar{k})$ for all *valid* re-execution vectors, it becomes less efficient to add the same $p_{\bar{n}_c}(\bar{k})$ for many different $\bar{k}$. Instead, it would be more efficient to (1) find the number of different $\bar{k}$ for which $p_{\bar{n}_c}(\bar{k})$ is the same, and (2) multiply this number with $p_{\bar{n}_c}(\bar{k})$ and add the product to the sum that evaluates the LoC. To achieve this, we introduce the new notations $\tilde{n}_c$ and $\tilde{k}$ to represent the reduced format of $\bar{n}_c$ and $\bar{k}$, respectively.

A distribution $\bar{n}_c$ which contains a number of execution segments which have the same size can be expressed in a reduced format $\tilde{n}_c$ by only including the execution segments that have different size, and then for each of these segments use an index to indicate how many execution segments have that particular size. Thus, the reduced format of a distribution $\bar{n}_c$ can be expressed as $\tilde{n}_c = [ES_1^{m_1}, ES_2^{m_2} ... ES_q^{m_q}]$, where $ES_i^{m_i}$ is used to indicate that there are m_i execution segments of size ES_i. Important to note is that the size of $\tilde{n}_c$ is q, while the size of $\bar{n}_c$ is n_c and $q \leq n_c$. In fact, q is equal to n_c if all the execution segments for a given distribution $\bar{n}_c$ have a different size. When q is equal to n_c, then all the indices m_i in $\tilde{n}_c$ are equal to one. However, when a number of execution segments have the same size, then q is always lower than n_c.

For a distribution that is expressed in a reduced format $\tilde{n}_c$, we can associate a re-execution vector that is also expressed in a reduced format. We denote the reduced format of a re-execution vector with $\tilde{k}$. The size of $\tilde{k}$ is equal to the size of $\tilde{n}_c$, and each $k_i \in \tilde{k}$ denotes the number of re-executions for any of the m_i execution segments with size ES_i given in $\tilde{n}_c$. Using the reduced format for a given distribution and a re-execution vector, *i.e.* $\tilde{n}_c$ and $\tilde{k}$, the completion time can be expressed with the following equation:

$$^{\tilde{n}_c}t_{\tilde{k}} = T + n_c \times \tau + \sum_{i=1}^{q} (ES_i + \tau) \times k_i \tag{8.13}$$

Note that Eq. (8.13) is similar to Eq. (8.1). However, the difference between Eq. (8.13) and Eq. (8.1) is that in Eq. (8.13) the sum goes from $i = 1$ to q (the size of $\tilde{n}_c$), while in Eq. (8.13) the sum goes from $i = 1$ to n_c (the size of $\bar{n}_c$).

The advantage of using the reduced format notations $\tilde{n}_c$ and $\tilde{k}$ is that for a given $\tilde{n}_c$ each re-execution vector represented with $\tilde{k}$ will result in a unique completion time. Next, we show how to compute the probability distribution function when the reduced format notations are used.

To express the probability distribution function with the new notations, *i.e* $\tilde{n}_c$ and $\tilde{k}$, we need to carefully consider what the notation $p_{\tilde{n}_c}(\tilde{k})$ represents. According to the definition, the probability distribution function provides the probability that a job completes at a given time, and therefore $p_{\tilde{n}_c}(\tilde{k})$ represents the probability that a job completes at a time ${}^{\tilde{n}_c}t_{\tilde{k}}$. When $\tilde{k}$ is used, the completion time is unique, but there are different number of cases that would result in the same completion time. To clarify this, let us consider that given are $\tilde{n}_c$ and $\tilde{k}$, and further consider that only the element k_i in $\tilde{k}$ is non-zero, while $k_j = 0 \ \forall \ j \neq i$. For the given example, the completion time includes k_i re-executions of an execution segment that has the size of ES_i. However, there may be m_i different execution segments that are of size ES_i (the i-th element in $\tilde{n}_c$ is $ES_i^{m_i}$), and therefore the k_i re-executions may occur as a result of erroneous execution of any of the m_i different execution segments of size ES_i. This implies that there are multiple cases such that a job completes after an execution segment of size ES_i has been re-executed k_i times. The number of combinations that k_i re-executions come from m_i different execution segments is given as $\binom{m_i+k_i-1}{k_i}$. Thus, the probability distribution function, given the notations $\tilde{n}_c$ and $\tilde{k}$, can be expressed with the following equation:

$$p_{\tilde{n}_c}(\tilde{k}) = {P_T}^2 \prod_{i=1}^{q} \binom{m_i + k_i - 1}{k_i} (1 - P_{e_i})^{k_i} \tag{8.14}$$

Finally, to compute the LoC, it is required to obtain all $\tilde{k}$ that result in a completion time lower or equal to the deadline. To achieve this, the function $f(\bar{k}, i)$ (Eq. (8.8)) can be used, where instead of $\bar{k}$, we use $\tilde{k}$ as the input argument, and the size of $\tilde{k}$, *i.e.* q, is used as the *update* index. Hence, the LoC is computed as follows. First, $\tilde{k}$ is initialized to zero ($k_i = 0 \ \forall \ k_i \in \tilde{k}$), and then the following steps are repeated until $\tilde{k} \neq \emptyset$: (1) $p_{\tilde{n}_c}(\tilde{k})$ is evaluated according to Eq. (8.14) and added in the sum that calculates the LoC, and (2) $\tilde{k} = f(\tilde{k}, q)$. Observe that by using the reduced format of both the distribution and the re-execution vector, *i.e.* $\tilde{n}_c$ and $\tilde{k}$, the function $f(\bar{k}, i)$ will be used more efficiently as it will only generate vectors $\tilde{k}$ that result in unique instances of the completion time. This speeds up the computation of the LoC.

8.4. EXHAUSTIVE SEARCH

In the previous section, we showed how to evaluate the LoC with respect to a given deadline for a given distribution of n_c checkpoints. However, the

problem we aim to solve is to obtain the optimal distribution, for a given number of checkpoints n_c, which results in the maximal LoC. As one way to obtain the optimal distribution of n_c checkpoints is to evaluate the LoC for all possible distributions of the given n_c checkpoints, in this section, we review an exhaustive search method that finds the optimal distribution that results in the maximal LoC. Further, we show that it is possible to speed up this method by avoiding exploration of distributions which are permutation of each other.

While a trivial exhaustive search approach can always find the optimal solution, the major drawback with any trivial exhaustive search algorithm is the complexity. For this particular problem, the complexity is directly proportional to the number of possible distributions of n_c checkpoints, and this number increases exponentially with n_c. However, in the previous section, we showed that the LoC for any distribution which is a permutation of a given distribution $\bar{n}_c$ will be the same as the LoC obtained for $\bar{n}_c$. This implies that there is no need to explore all possible distributions to find the optimal distribution that results in the maximal LoC. For that reason, in this section, we discuss a method, namely Exhaustive Search (EXS), that finds the optimal distribution by only evaluating the LoC for distributions $\bar{n}_c$ which are not permutations of each other. In particular, we show how to obtain these distributions, and we determine the number of different distributions that need to be explored with this method. Furthermore, we compare the complexity of the EXS method with a trivial exhaustive search method that explores all possible distributions of a given number of checkpoints n_c.

The complexity of the trivial exhaustive search method is directly proportional to the number of *all* possible distributions of n_c checkpoints in a job with a processing time T, and this number is evaluated with the following equation:

$$\binom{T-1}{n_c-1} = \frac{(T-1)!}{(T-n_c)!(n_c-1)!} \tag{8.15}$$

Assuming that a checkpoint can be taken at any integer time unit, for a job with processing time T, expressed in time units, there are T different points where a checkpoint can be taken. Since one checkpoint must be taken at the end of the job, *i.e.* at time unit T, the rest of the $n_c - 1$ checkpoints can be taken at any of the remaining $T - 1$ time units. Therefore, the number of combinations of $n_c - 1$ checkpoints over $T - 1$ time points represents the amount of *all* possible distributions of n_c checkpoints as it is expressed with Eq. (8.15).

In Table 8.1, we report the number of *all* possible distributions of n_c checkpoints in a job with a processing time $T = 1000$ t.u. at various instances of n_c. As can be seen from Table 8.1, the number of possible distributions grows rapidly with n_c.

n_c	$\binom{T-1}{n_c-1}$
2	999
3	498501
4	165668499
5	41251456251
6	8209039793949
7	1359964259197551
8	192920644197595449
9	23922159880501835676
10	2634095604619702128324

Table 8.1.: Number of *all* possible distributions of n_c checkpoints in a job with a processing time $T = 1000$ t.u.

While the trivial exhaustive search approach explores *all* possible distributions of n_c checkpoints, the EXS method does not need to evaluate the LoC for *all* distributions, and instead, it should only explore distributions which are not permutations of each other. To achieve this goal, it is sufficient to explore distributions $\bar{n}_c$ which satisfy the following condition:

$$ES_i \leq ES_j \; \forall i \leq j \tag{8.16}$$

An important consequence that follows from the condition expressed in Eq. (8.16) is that the size of the shortest execution segment in a distribution $\bar{n}_c$, *i.e.* ES_1, can never be larger than $\left\lfloor \frac{T}{n_c} \right\rfloor$, where T represents the processing time of the job and n_c represents the number of checkpoints that are to be distributed. Next, we elaborate how to obtain all those distributions that satisfy the condition in Eq. (8.16).

To obtain the different distributions $\bar{n}_c$, we use the function $next(\bar{n}_c, i)$, given in Eq. (8.19), which for a given distribution $\bar{n}_c$ and an *update* index i, generates the next distribution to be explored. The function $next(\bar{n}_c, i)$ first computes a tentative distribution $\bar{n}_c^N$ (Eq. (8.18)) by incrementing the size of the i-th execution segments by one ($ES_i = ES_i + 1$), and assigning the same incremented value to all execution segments with index in the range $[i+1, n_c - 1]$. The size of the last execution segment, *i.e.* ES_{n_c}, is computed such that the sum of all the execution segments is the same as the given processing time of the job T (see Eq. (8.17)). If the tentative distribution $\bar{n}_c^N$ satisfies the condition in Eq. (8.16), then the function returns $\bar{n}_c^N$ as output.

Otherwise, the function steps in a recursive call by decrementing the update index i, *i.e.* $next(\bar{n}_c, i-1)$. If the function is called with an update index $i = 0$, the function returns an empty set, meaning that $\bar{n}_c$ is the last distribution to be explored.

$$ES_{n_c} = T - \sum_{j=0}^{i-1} ES_j - (n_c - i) \times (ES_i + 1) \tag{8.17}$$

$$\bar{n}_c^N = [ES_1, ES_2, ...ES_{i-1}, ES_i + 1, ES_i + 1...ES_{n_c}] \tag{8.18}$$

$$next(\bar{n}_c, i) = \begin{cases} \bar{n}_c^N, \text{ iff Eq. (8.16) holds} \\ next(\bar{n}_c, i-1), \text{ iff Eq. (8.16) does not hold} \\ \emptyset, \text{ iff } i = 0 \end{cases} \tag{8.19}$$

When the function is used, the function is always invoked with the current distribution $\bar{n}_c$ that is being explored and $n_c - 1$ as an update index. The initial distribution is constructed by assigning the value of 1 (the shortest size of an execution segment) to all execution segments except for the last one. The size of the last execution segment is computed such that the sum of all execution segments is equal to the processing time of the job T. In other words, the initial distribution is presented with the following vector $\bar{n}_c = [1, 1...1, T - n_c + 1]$.

By using this function, the number of distributions which are explored with the EXS method is reduced when compared to the number of all possible distributions given with Eq. (8.15). Next, we elaborate how to compute $r_{n_c}[T]$, the number of distributions of n_c checkpoints in a job with a processing time T, which are explored with the EXS method.

We denote with $r_{n_c}^i[T]$ the number of distributions of n_c checkpoints in a job with a processing time T, where the size of the first execution segment, *i.e.* ES_1, is fixed to a value i. Due to the consequence which followed from Eq. (8.16), the size of the first execution segment cannot be larger than $\left\lfloor \frac{T}{n_c} \right\rfloor$. Hence, we compute $r_{n_c}[T]$ with the following expression:

$$r_{n_c}[T] = \sum_{i=1}^{\left\lfloor \frac{T}{n_c} \right\rfloor} r_{n_c}^i[T] \tag{8.20}$$

When the size of the first execution segment in a distribution $\bar{n}_c$ is fixed to a value i, $r_{n_c}^i[T]$ can be obtained by counting how many distributions of $n_c - 1$ checkpoints in a job with a processing time $T - i$ exist, such that the shortest execution segment among the $n_c - 1$ execution segments has to be larger or equal to i. Observe that when distributing $n_c - 1$ checkpoints in a

job with a processing time $T - i$, the shortest execution segment cannot be larger than $\frac{T-i}{n_c-1}$ (the consequence of Eq. (8.16)). Given this, the following recurrence equation can be used to calculate $r^i_{n_c}[T]$:

$$r^i_{n_c}[T] = \sum_{j=i}^{\lfloor \frac{T-i}{n_c-1} \rfloor} r^j_{n_c-1}[T-i] \tag{8.21}$$

The initial condition of the recurrence equation given with Eq. (8.21) is expressed as:

$$r^i_3[T] = \left\lfloor \frac{T-i}{2} \right\rfloor - i + 1 \tag{8.22}$$

Namely, the expression in Eq. (8.22) is derived from the fact that there exists only a single distribution of two checkpoints in a job with a processing time T such that the size of the first execution segment is fixed to a given value. When the size of the first execution segment in a distribution of two checkpoints is fixed to a value i, the size of the second execution segment is directly calculated by subtracting the value i from the processing time of the job. Following the recurrence equation, Eq. (8.21), we get:

$$\begin{aligned} r^i_3[T] &= \sum_{j=i}^{\lfloor \frac{T-i}{2} \rfloor} r^j_2[T-i] \\ r^i_3[T] &= \sum_{j=i}^{\lfloor \frac{T-i}{2} \rfloor} 1 \\ r^i_3[T] &= \left\lfloor \frac{T-i}{2} \right\rfloor - i + 1 \end{aligned} \tag{8.23}$$

In Table 8.2, we show $r_{n_c}[T]$, obtained by using Eq. (8.20) and Eq. (8.21), for different values of n_c and a job with a processing time $T = 1000$ t.u.

To show the reduced complexity of the EXS method over the trivial exhaustive search approach, we compare the results presented in Table 8.1 and Table 8.2. As shown in Table 8.1 and Table 8.2, the complexity of the EXS method is significantly lower than the complexity of the trivial exhaustive search approach. For example, we see that for $n_c = 10$, the number of all possible distributions is $\approx 2 \times 10^{21}$ (observe Table 8.1 for $n_c = 10$). However, by excluding the distributions which are permutation of each other, the number of different distributions that are explored by the EXS method is $\approx 1 \times 10^{15}$ (observe Table 8.1 for $n_c = 10$). From this example, it should be evident that by excluding the redundant distributions the complexity of the EXS method is reduced tremendously in comparison to the trivial exhaustive search. Still,

n_c	$r_{n_c}[T]$
3	83333
4	6965278
5	350697875
6	11835956777
7	287302124354
8	5274078114658
9	76037051194142
10	886745696653253

Table 8.2.: Number of distributions of n_c checkpoints, explored with the EXS method, for a job with a processing time $T = 1000$ t.u.

the complexity of the EXS method is very high. For that reason, we developed the Clustered Checkpointing (CC) method to find the distribution that results in the highest LoC for a given n_c.

8.5. CLUSTERED CHECKPOINTING

In this section, we present the Clustered Checkpointing (CC) method which distributes a given number of checkpoints such that the LoC is maximized. The CC heuristic explores only distributions $\bar{n}_c$ which are made out of clusters of execution segments, where all the execution segments that belong to the same cluster have the same size and the maximum number of clusters in $\bar{n}_c$ is set to three.

Given the following inputs: a processing time of a job T, a deadline D, a checkpointing overhead τ, a probability P_T that no errors occur in a processing node within an interval of length T, and a number of checkpoints n_c, the CC method outputs the distribution $\bar{n}_c^*$ which provides the highest LoC among all distributions $\bar{n}_c$ which are explored with this method. A pseudo-code algorithm for the CC method is presented in Figure 8.1. As shown in Figure 8.1, the method consists of two loops: an inner loop (see steps 3-13 in the figure) and an outer loop (see steps 2-14 in the figure). The inner loop performs a total of $n_c - 1$ iterations, and in each iteration, first, it constructs a distribution $\bar{n}_c$, and then, it evaluates the LoC with respect to the given deadline for the constructed $\bar{n}_c$ and compares it with the most recently computed Λ_{max} which holds the initial value zero (see step 1 in Figure 8.1). In the first

ALGORITHM CC(n_c, T, τ, P_T, D)

1. $\Lambda_{max} = 0$
2. **for** $L = 1$ **to** $\left\lfloor \frac{T}{n_c} \right\rfloor$ **do**
3. **for** $i = 1$ **to** $n_c - 1$ **do**
4. $\bar{n}_c[i] = L$
5. $\hat{T} = T - i \times L$
6. $\hat{n}_c = n_c - i$
7. $\bar{n}_c[j] = \left\lfloor \frac{\hat{T}}{\hat{n}_c} \right\rfloor \ \forall\, j \in [i+1, i+\hat{n}_c - (\hat{T} \ \textbf{mod}\ \hat{n}_c)]$
8. $\bar{n}_c[k] = \left\lceil \frac{\hat{T}}{\hat{n}_c} \right\rceil \ \forall\, k \in [i+1+\hat{n}_c - (\hat{T} \ \textbf{mod}\ \hat{n}_c),\ n_c]$
9. **if** $(\Lambda_{\bar{n}_c}(D) \geq \Lambda_{max})$ **then**
10. $\Lambda_{max} = \Lambda_{\bar{n}_c}(D)$
11. $\bar{n}_c^* = \bar{n}_c$
12. **end if**
13. **end for**
14. **end for**
15. **return** $\bar{n}_c^*,\ \Lambda_{max}$

Figure 8.1.: Flow chart of the Clustered Checkpointing method

iteration ($i = 1$), the inner loop starts by setting the size of the first execution segment ES_1 to a given value L (step 4 in Figure 8.1), while the size of the remaining execution segments is obtained by "equally" distributing the rest of the checkpoints $\hat{n}_c = n_c - 1$ over the remaining processing time, *i.e.* $\hat{T} = T - L$ (observe that $\hat{T}$ and $\hat{n}_c$ are evaluated in step 5 and 6 in Figure 8.1). Since $\hat{T}$ is not necessarily a multiple of $\hat{n}_c$ and only integer values are allowed for the size of the execution segments, the following scheme is used to distribute the remaining $\hat{n}_c$ checkpoints. From the remaining $\hat{n}_c$ execution segments, the size of the first $\hat{n}_c - (\hat{T}$ **modulo** $\hat{n}_c)$ execution segments is set to $\left\lfloor \frac{\hat{T}}{\hat{n}_c} \right\rfloor$ (see step 7 in Figure 8.1), and for the rest of the execution segments the size is set to $\left\lceil \frac{\hat{T}}{\hat{n}_c} \right\rceil$ (see step 8 in Figure 8.1). Therefore, $\bar{n}_c$ consists of no more than three clusters (two clusters if $\hat{T}$ is a multiple of $\hat{n}_c$) which are ordered in non-decreasing order based on the size of the execution segments, and the shortest size of the execution segments is L. Such ordering is needed to prevent exploration of distributions $\bar{n}_c$ which may be permutation of each other. Finally, for the constructed distribution $\bar{n}_c$, the LoC is evaluated and compared with the most recently computed maximum (see step 9 in Figure 8.1). If $\Lambda_{\bar{n}_c}(D) > \Lambda_{max}$, it means that the distribution $\bar{n}_c$ provides a higher LoC than Λ_{max}, and therefore Λ_{max} is set to $\Lambda_{\bar{n}_c}(D)$ (step 10 in Figure 8.1) and $\bar{n}_c^*$ is set to $\bar{n}_c$ (step 11 in Figure 8.1).

In the i-th iteration, ES_i is set to the given value L (step 4 in Figure 8.1), while the size of the remaining $\hat{n}_c = n_c - i$ execution segments is obtained by "equally" distributing the rest of the checkpoints $\hat{n}_c$ over the remaining processing time $\hat{T} = T - i \times L$ (steps 5-8 in Figure 8.1). The LoC for the constructed distribution is evaluated and compared against Λ_{max} (step 9 in Figure 8.1).

The outer loop is used to control the value of L which is utilized in the inner loop (see step 2 in Figure 8.1). To ensure that the non-decreasing ordering of the clusters in the explored distributions $\bar{n}_c$ is preserved, the size of the shortest execution segment cannot be larger than $\left\lfloor \frac{T}{n_c} \right\rfloor$. Therefore, the outer loop performs a total of $\left\lfloor \frac{T}{n_c} \right\rfloor$ iterations, *i.e.* $L \in [1, \left\lfloor \frac{T}{n_c} \right\rfloor]$.

Finally, the CC method outputs $\bar{n}_c^*$ after exploring a total of $\left\lfloor \frac{T}{n_c} \right\rfloor \times (n_c - 1)$ different distributions.

Illustration of how the different distributions explored by the CC method are constructed is provided in the following example.

8.5.1. EXAMPLE

The purpose of this example is to illustrate the different distributions which are explored with the CC method, and to clarify the notion of clusters. Few

sample distributions are presented, and for each distribution the notion of clusters is illustrated.

Example 1. Given the following set of parameters:

- processing time $T = 24$ t.u., and
- a number of checkpoints $n_c = 5$,

we present how the different distributions which are explored with the CC method are constructed. Observe that for this example, it is sufficient to only have the two input parameters, *i.e.* the processing time T and the number of checkpoints n_c, since the rest of the input parameters used by the CC method, *i.e.* the deadline D, the checkpointing overhead τ, and the probability P_T that no errors occur in a processing node within an interval of length T, are used to compute the LoC with respect to the given deadline for each explored distribution. Since the focus of this example is to show how the different distributions are constructed, only the two input parameters, stated previously, are needed.

As shown in the previous section, the CC method only explores distributions $\bar{n}_c$ where the execution segments are ordered in a non-decreasing order. The reason for this is to avoid distributions which are permutation of each other. In such ordering, there exists an upper bound on the size of the shortest execution segment used in the distribution, and as shown in the previous section, this bound is obtained as $\left\lfloor \frac{T}{n_c} \right\rfloor$. For the given inputs, in this example, the size of the shortest execution segment cannot be larger than $\left\lfloor \frac{24}{5} \right\rfloor = 4$. On the other hand, due to the assumption that a checkpoint can be taken at every integer time unit, the shortest size of an execution segment is one. Therefore, for this example, the variable L used in the outer loop of the CC method iterates through the range of values from 1 to 4 (observe step 2 in Figure 8.1). Next, we illustrate the distributions $\bar{n}_c$ that are constructed, given that the variable L has reached the value $L = 3$. The distributions are illustrated in Figure 8.2. Since the inner loop in the CC method performs $n_c - 1$ iterations (observe step 3 in Figure 8.1) and in each iteration it constructs one distribution, for a given value of L only $n_c - 1$ different distributions are constructed. For this example, since $n_c = 5$, only 4 different distributions are constructed for $L = 3$. The first distribution is illustrated in Figure 8.2 (a), and it is constructed as follows. In the first iteration ($i = 1$) of the inner loop, the size of the first execution segment is set to 3 (observe step 4 in Figure 8.1). This implies that the rest of the processing time $\hat{T}$ is 21 t.u. (see step 5 in Figure 8.1), and the number of the remaining checkpoints $\hat{n}_c$ that should be distributed is 4 (see step 6 in Figure 8.1). Next, the CC method tries to evenly distribute the remaining four checkpoints over the rest of the processing time $\hat{T} = 21$ t.u. Since 21 is not

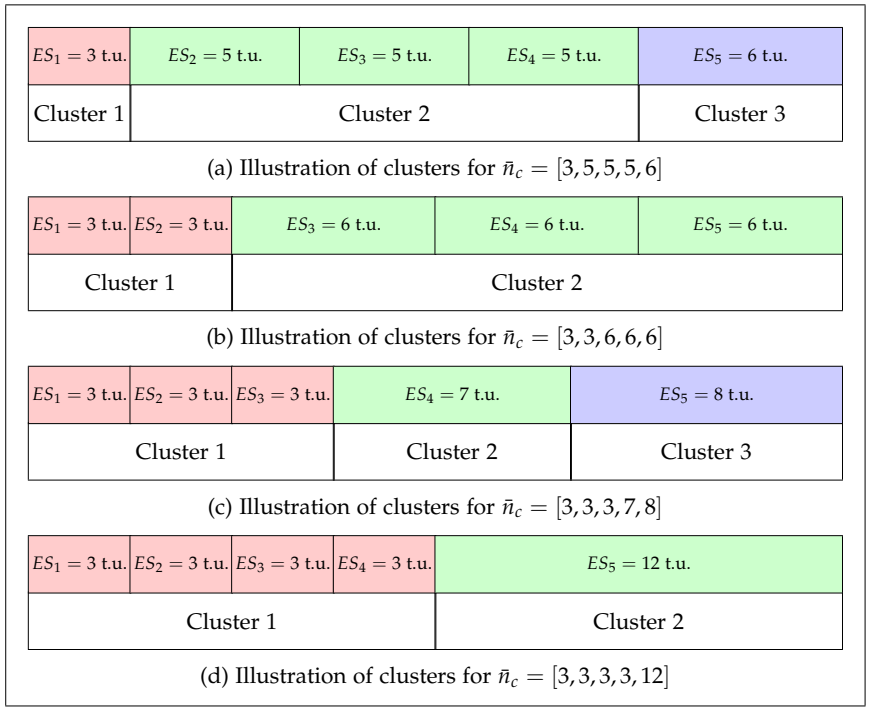


Figure 8.2.: Distributions explored by the CC method for the given inputs $L = 3,\ T = 24$ t.u., and $n_c = 5$

a multiple of 4, distributing the checkpoints evenly is not possible. Instead, three execution segments need to be of size $\left\lfloor \frac{21}{4} \right\rfloor = 5$, and one execution segment needs to be of size $\left\lceil \frac{21}{4} \right\rceil = 6$. To preserve the non-decreasing ordering in the distribution, the CC method first assigns $ES_j = 5$ t.u. $\forall\ j\ \in [2,4]$ (see step 7 in Figure 8.1), and then it assigns $ES_k = 6$ t.u. for $k = 5$ (see step 8 in Figure 8.1). At this point the CC method has constructed the first distribution $\bar{n}_c = [3,5,5,5,6]$, and this distribution is illustrated in Figure 8.2 (a). As shown in Figure 8.2 (a), the distribution $\bar{n}_c = [3,5,5,5,6]$ consists of three clusters, namely Cluster 1, Cluster 2, and Cluster 3, where all the execution segments that belong to the same cluster have the same size, and they are marked with the same color. Cluster 1 contains only one execution segment of size 3 t.u. and it is marked with red (observe ES 1 in Figure 8.2 (a)), Cluster 2 contains three execution segments of size 5 t.u. and these execution segments are marked with green (observe ES 2, ES 3 and ES 4 in Figure 8.2 (a)), and

Cluster 3 contains a single execution segment of size 6 t.u. and it is marked with blue (observe ES_5 in Figure 8.2 (a)).

The second distribution is constructed during the second iteration ($i = 2$) of the inner loop of the CC method (step 3 in Figure 8.1), and it is illustrated in Figure 8.2 (b). The distribution is constructed as follows. First, the size of the second execution segment is set to 3 (see step 4 in Figure 8.1). At this point the size of the first execution segment remains unchanged since the previous iteration, *i.e.* the size of the first execution segments is 3 t.u.. Since the size of the first two execution segments is fixed to 3 t.u., the rest of the processing time $\hat{T}$ is $24 - 2 \times 3 = 18$ t.u. (step 5 in Figure 8.1) and the number of remaining checkpoints $\hat{n}_c$ is set to 3 (step 6 in Figure 8.1). The size of the remaining three execution segments is obtained by evenly distributing the remaining three checkpoints over the rest of the processing time. Since 18 is a multiple of three, the size of the remaining three execution segments is set to 6 t.u., and therefore the CC method assigns $ES_j = 6$ t.u. $\forall j \in [3, 5]$ (step 7 in Figure 8.1). Therefore, the constructed distribution during the second iteration is $\bar{n}_c = [3, 3, 6, 6, 6]$, and it is illustrated in Figure 8.2 (b). As shown in Figure 8.2 (b), $\bar{n}_c = [3, 3, 6, 6, 6]$ consists of two clusters, *i.e.* Cluster 1 which contains two execution segments of size 3 t.u. (observe ES_1 and ES_2 in Figure 8.2 (b)), and Cluster 2 which contains three execution segments of size 6 t.u. (observe ES_3, ES_4 and ES_5 in Figure 8.2 (b)).

The third distribution is constructed during the third iteration of the inner loop, and it is illustrated in Figure 8.2 (c). First, the size of the third execution segment is set to 3 t.u., and hence the size of the first three execution segments is set to 3 t.u.. The size of the remaining two execution segments is obtained by evenly distributing the remaining two checkpoints over the rest of the processing time $\hat{T} = 24 - 3 \times 3 = 15$ t.u. Since 15 is not a multiple of two, the CC method sets the size of one execution segment to 7 t.u. and the size of the other execution segment to 8 t.u., and thus it constructs the distribution $\bar{n}_c = [3, 3, 3, 7, 8]$ which is illustrated in Figure 8.2 (c). As shown in Figure 8.2 (c), $\bar{n}_c = [3, 3, 3, 7, 8]$ consists of three clusters. Cluster 1 contains three execution segments of size 3 t.u. (observe ES_1, ES_2 and ES_3 marked with red in Figure 8.2 (c)), Cluster 2 contains one execution segment of size 7 t.u. (observe ES_4 marked with green in Figure 8.2 (c)), and Cluster 3 contains one execution segment of size 8 t.u. (observe ES_5 marked with blue in Figure 8.2 (c)).

Finally, the fourth constructed distribution is $\bar{n}_c = [3, 3, 3, 3, 12]$, and it is illustrated in Figure 8.2 (d). As shown in Figure 8.2 (d), $\bar{n}_c = [3, 3, 3, 3, 12]$ consists of two clusters, *i.e.* Cluster 1 which contains four execution segments of size 3 t.u. (observe ES_1, ES_2, ES_3 and ES_4 in Figure 8.2 (d)), and Cluster 2 which contains one execution segments of size 12 t.u. (observe ES_5 in Figure 8.2 (d)).

Scenario A	Scenario B	Scenario C
$T = 100$ t.u.	$T = 1000$ t.u.	$T = 1000$ t.u.
$D = 150$ t.u.	$D = 1300$ t.u.	$D = 1300$ t.u.
$\tau = 2$ t.u.	$\tau = 10$ t.u.	$\tau = 20$ t.u.
$P_T = 0.99999$	$P_T = 0.99999$	$P_T = 0.99999$

Table 8.3.: Input scenarios

8.6. EXPERIMENTAL RESULTS

The objective of the experiments presented in this section is as follows. First, we study the impact on the LoC when a checkpoint can only be taken at an integer time unit. Second, given that a checkpoint can only be taken at an integer time unit, we evaluate to what extent the proposed heuristic (the CC method) is able to find the optimal solution which is obtained by using the Exhaustive Search (EXS) method. Third, we evaluate which of the two schemes, *i.e.* equidistant and non-equidistant checkpointing, provides the highest LoC. For that reason, we present results for the following two experiments:

- P1: for various number of checkpoints compare the LoC for the exact equidistant checkpointing scheme (EEQC) and the LoC for the limited equidistant checkpointing scheme (LEQC) where the checkpoints are "evenly" distributed while applying the limitation that a checkpoint can only be taken at an integer time unit;

- P2: given that a checkpoint can only be taken at an integer time unit, for various number of checkpoints compare the LoC for the equidistant and the non-equidistant checkpointing scheme.

For the given experiments P1 and P2, we use the input scenarios given in Table 8.3. Each scenario is defined with the following parameters: the processing time of the job T, the deadline D, the checkpointing overhead τ, and the probability P_T that no errors occur in a processing node within an interval of length T.

As the goal in P1 and P2 is to evaluate the LoC for different checkpointing schemes at various number of checkpoints, important to note is that for each of the scenarios presented in Table 8.3 the range of values for the number of checkpoints n_c is different. This follows from the fact that if n_c is sufficiently high, the deadline will be violated ($\Lambda_{\bar{n}_c}(D) = 0$). Hence, there exists an upper bound to determine the range of values for n_c. The upper bound of n_c, in all scenarios, is obtained from the condition that the best case execution time, *i.e.*

time required for the job to complete when no errors occur $T + n_c \times \tau$, should not violate the deadline. Therefore, for Scenario A $n_c \in [2, 25]$, for Scenario B $n_c \in [2, 30]$ and for Scenario C $n_c \in [2, 15]$.

Important to note is that evaluating the LoC usually results in numbers which are very close to 1, *i.e.* $\Lambda_{\bar{n}_c}(D) \approx 1$. Since the values $\Lambda_{\bar{n}_c}(D)$ are very close to 1, the difference $\bar{\Lambda}_{\bar{n}_c}(D) = 1 - \Lambda_{\bar{n}_c}(D)$ results in numbers that are more convenient to present by using scientific notation. Therefore, when we present the results for P1 and P2 for the different scenarios given in Table 8.3, we show the values $\bar{\Lambda}_{\bar{n}_c}(D)$. One observes that lower values for $\bar{\Lambda}_{\bar{n}_c}(D)$ are better, *i.e.* $\Lambda_{\bar{n}_c}(D)$ is higher. Next, we present the results for the two experiments P1 and P2.

For P1, the results are summarized in Tables 8.4–8.6 for Scenario A, B, and C, respectively. The results presented in Tables 8.4–8.6 show $\bar{\Lambda}_{\bar{n}_c}(D)$ obtained from the two checkpointing schemes EEQC and LEQC at various number of checkpoints n_c. Additionally, in Tables 8.4–8.6, the distribution of checkpoints $\bar{n}_c$ for the LEQC is presented for each n_c, and for convenience we use the reduced format of the distribution $\tilde{n}_c$. For each n_c, we highlight which of the two schemes provides the highest LoC (the result is marked with bold). With these results, we study the impact on the LoC when a checkpoint can only be taken at an integer time unit.

The EEQC scheme distributes the n_c checkpoints evenly without any limitation on the checkpoint placement, and considers that all execution segments are of the same size $\frac{T}{n_c}$. On the other hand, the LEQC scheme distributes the n_c checkpoints as even as possible, and it considers that a checkpoint can only be taken at an integer time unit. Therefore, when the processing time T, given in time units, is not a multiple of the number of checkpoints n_c, the LEQC results in a distribution of checkpoints $\bar{n}_c$ where the execution segments are divided into two clusters, where all the execution segments that belong to the first cluster have the size $\left\lfloor \frac{T}{n_c} \right\rfloor$, and all the execution segments that belong to the second cluster have the size $\left\lceil \frac{T}{n_c} \right\rceil$. Observe that if the processing time T is a multiple of the number of checkpoints n_c, then both schemes evaluate the LoC for the same distribution of checkpoints $\bar{n}_c$ where the size of all the execution segments is $\frac{T}{n_c}$, and therefore in such case, the LoC will be the same for both schemes. Next, we discuss the results presented in Tables 8.4–8.6.

From the results shown in Tables 8.4–8.6, we draw the conclusion that the limitation on the checkpoint placement, *i.e.* a checkpoint can only be taken at an integer time unit, affects the LoC only in the following two cases: (1) the processing time T is not a multiple of the number of checkpoints n_c, and (2) for the given number of checkpoints n_c, at least one re-execution can take place without violating the deadline. Since for Scenario C, for any $n_c \in [2, 15]$ it is not possible to have even a single re-execution without violating the deadline,

the results for all n_c are the same, and therefore the LoC is not affected by the limitation on the checkpoint placement. In most of the cases where the LoC is affected, the limitation on the checkpoint placement degrades the LoC (observe the results in Table 8.6). However, there are exceptional cases where the limitation on the checkpoint placement may lead to an improved LoC (observe the results for $n_c = 18$ and $n_c = 22$ in Table 8.4). Next, we elaborate on these two exceptional cases.

For Scenario A and $n_c = 18$, the size of the execution segments for the EEQC scheme is $\frac{100}{18}$ and with such size it is possible to re-execute a only single execution segment without violating the deadline (this is evaluated with Eq. (4.28) in Section 4.1.4 in Chapter 4). However, for the same inputs the LEQC scheme results in a distribution where eight execution segments are of size equal to $\left\lfloor \frac{100}{18} \right\rfloor = 5$ t.u., and 10 execution segments are of size equal to $\left\lceil \frac{100}{18} \right\rceil = 6$ t.u. Furthermore, any of the eight execution segments can be re-executed twice without violating the deadline. Therefore, the LoC obtained from the LEQC scheme will be higher as it includes the probability that some execution segments may be re-executed twice.

For Scenario A and $n_c = 22$, the size of the execution segments for the EEQC scheme is $\frac{100}{22}$ and with such size any re-execution violates the deadline (the maximum number of re-executions is evaluated as zero using Eq. (4.28)). On the other hand, the LEQC scheme results in a distribution where 10 execution segments are of size $\left\lfloor \frac{100}{18} \right\rfloor = 4$ t.u. and 12 execution segments are of size $\left\lceil \frac{100}{18} \right\rceil = 5$ t.u. Furthermore, any of the 10 execution segments of size 4 t.u.can be re-executed once without violating the deadline. Therefore, the LoC obtained from the LEQC scheme is higher. From these two cases we conclude that the limitation on the checkpoint placement will have a positive effect on the LoC (higher LoC) only when the following condition holds:

$$\left\lfloor \frac{D - T - n_c \times \tau}{\left\lfloor \frac{T}{n_c} \right\rfloor} \right\rfloor > \left\lfloor \frac{D - T - n_c \times \tau}{\frac{T}{n_c}} \right\rfloor \tag{8.24}$$

The condition in Eq. (8.24) compares if the number of re-executions of an execution segment of size $\left\lfloor \frac{T}{n_c} \right\rfloor$ is greater than the number of re-executions of an execution segment of size $\frac{T}{n_c}$. Only when this condition is satisfied, the LEQC scheme will outperform the EEQC scheme. Next, we present the results for the second experiment P2.

For P2, the results are summarized in Tables 8.7–8.9, for Scenario A, B, and C, respectively. These results show $\bar{\Lambda}_{\bar{n}_c}(D)$ (along with $\bar{n}_c$) obtained from the three methods, *i.e.* LEQC, CC, and EXS for different values of n_c. For each

n_c	EEQC	LEQC	n_c	EEQC	LEQC
2	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [50^2]$	14	**1.632651e-15**	1.633438e-15 $\tilde{n}_c = [7^{12}, 8^2]$
3	**2.666651e-10**	2.666785e-10 $\tilde{n}_c = [33^2, 34^1]$	15	**1.611850e-15**	1.613358e-15 $\tilde{n}_c = [6^5, 7^{10}]$
4	**2.499987e-10**	**2.499987e-10** $\tilde{n}_c = [25^4]$	16	**1.593748e-15**	1.595102e-15 $\tilde{n}_c = [6^{12}, 7^4]$
5	**2.399988e-10**	**2.399988e-10** $\tilde{n}_c = [20^5]$	17	**1.577853e-15**	1.578638e-15 $\tilde{n}_c = [5^2, 6^{15}]$
6	**2.074068e-15**	**2.074778e-15** $\tilde{n}_c = [16^2, 17^4]$	18	2.111103e-10	**1.751996e-10** $\tilde{n}_c = [5^8, 6^{10}]$
7	**1.959179e-15**	1.959915e-15 $\tilde{n}_c = [14^5, 15^2]$	19	**2.105255e-10**	2.105992e-10 $\tilde{n}_c = [5^{14}, 6^5]$
8	**1.874996e-15**	1.875996e-15 $\tilde{n}_c = [12^4, 13^4]$	20	**2.099992e-10**	**2.099992e-10** $\tilde{n}_c = [5^{20}]$
9	**1.810696e-15**	1.811132e-15 $\tilde{n}_c = [11^8, 12^1]$	21	**2.095230e-10**	2.095992e-10 $\tilde{n}_c = [4^5, 5^{16}]$
10	**1.759997e-15**	**1.759997e-15** $\tilde{n}_c = [10^{10}]$	22	1.99999e-5	**1.200002e-5** $\tilde{n}_c = [4^{10}, 5^{12}]$
11	**1.719005e-15**	1.719437e-15 $\tilde{n}_c = [9^{10}, 10^1]$	23	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [4^{15}, 5^8]$
12	**1.685183e-15**	1.686429e-15 $\tilde{n}_c = [8^8, 9^4]$	24	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [4^{20}, 5^4]$
13	**1.656802e-15**	1.658078e-15 $\tilde{n}_c = [7^4, 8^9]$	25	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [4^{25}]$

Table 8.4.: Comparison of $\bar{\Lambda}_{n_c}(D)$ for different checkpointing schemes at different n_c values for Scenario A

n_c, we highlight which of the three methods provides the highest LoC (the result is marked with bold). Observe that since in Tables 8.7–8.9 we show $\bar{\Lambda}_{\bar{n}_c}(D)$, smaller values of $\bar{\Lambda}_{\bar{n}_c}(D)$ indicate a better result, *i.e.* higher $\Lambda_{\bar{n}_c}(D)$. The objective with these results is to show that (1) the proposed heuristic (the CC method) is able to find the optimal solution in most of the cases, and (2) the LoC is improved when non-equidistant checkpointing is used. Next, we discuss the results presented in Tables 8.7–8.9.

As can be seen from Tables 8.7–8.9, comparing the results from the EXS method with the results from the CC method shows that the CC method in most of the presented cases is able to find the optimal $\bar{n}_c^*$. The advantage of the CC method is that it finds the solution in significantly shorter time than the

n_c	EEQC	LEQC	n_c	EEQC	LEQC
2	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [500^2]$	17	**2.117639e-10**	2.117644e-10 $\tilde{n}_c = [58^3, 59^{14}]$
3	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [333^2, 334^1]$	18	**2.111103e-10**	2.111112e-10 $\tilde{n}_c = [55^8, 56^{10}]$
4	**2.499987e-10**	**2.499987e-10** $\tilde{n}_c = [250^4]$	19	**2.105255e-10**	2.105264e-10 $\tilde{n}_c = [52^7, 53^{12}]$
5	**2.399988e-10**	**2.399988e-10** $\tilde{n}_c = [200^5]$	20	**2.099992e-10**	**2.099992e-10** $\tilde{n}_c = [50^{20}]$
6	**2.333322e-10**	2.333325e-10 $\tilde{n}_c = [166^2, 167^4]$	21	**2.095230e-10**	2.095240e-10 $\tilde{n}_c = [47^8, 48^{13}]$
7	**2.285704e-10**	2.285706e-10 $\tilde{n}_c = [142^1, 143^6]$	22	**2.090901e-10**	2.090912e-10 $\tilde{n}_c = [45^{12}, 46^{10}]$
8	**2.249990e-10**	**2.249990e-10** $\tilde{n}_c = [125^8]$	23	**2.086949e-10**	2.086960e-10 $\tilde{n}_c = [43^{12}, 44^{11}]$
9	**2.222213e-10**	2.222215e-10 $\tilde{n}_c = [111^8, 112^1]$	24	**2.083326e-10**	2.083336e-10 $\tilde{n}_c = [41^8, 42^{16}]$
10	**2.199991e-10**	**2.199991e-10** $\tilde{n}_c = [100^{10}]$	25	**2.079993e-10**	**2.079993e-10** $\tilde{n}_c = [40^{25}]$
11	**2.181809e-10**	2.181811e-10 $\tilde{n}_c = [90^1, 91^{10}]$	26	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [38^{14}, 39^{12}]$
12	**2.166658e-10**	2.166663e-10 $\tilde{n}_c = [83^8, 84^4]$	27	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [37^{26}, 38^1]$
13	**2.153837e-10**	2.153839e-10 $\tilde{n}_c = [76^1, 77^{12}]$	28	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [35^8, 36^{20}]$
14	**2.142848e-10**	2.142855e-10 $\tilde{n}_c = [71^8, 72^6]$	29	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [34^{15}, 35^{14}]$
15	**2.133325e-10**	2.133332e-10 $\tilde{n}_c = [66^5, 67^{10}]$	30	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [33^{20}, 34^{10}]$
16	**2.124992e-10**	2.125000e-10 $\tilde{n}_c = [62^8, 63^8]$	31		

Table 8.5.: Comparison of $\bar{\Lambda}_{n_c}(D)$ for different checkpointing schemes at different n_c values for Scenario B

n_c	EEQC	LEQC	n_c	EEQC	LEQC
2	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [500^2]$	9	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [111^8, 112^1]$
3	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [333^2, 334^1]$	10	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [100^{10}]$
4	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [250^4]$	11	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [90^1, 91^{10}]$
5	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [200^5]$	12	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [83^8, 84^4]$
6	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [166^2, 167^4]$	13	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [76^1, 77^{12}]$
7	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [142^1, 143^6]$	14	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [71^8, 72^6]$
8	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [125^8]$	15	**1.99999e-5**	**1.99999e-5** $\tilde{n}_c = [66^5, 67^{10}]$

Table 8.6.: Comparison of $\bar{\Lambda}_{n_c}(D)$ for different checkpointing schemes at different n_c values for Scenario C

EXS method. The reason that no results are reported for the EXS method for n_c values larger than five, in Tables 8.8–8.9, is that EXS is very time-consuming which is due to the large number of distributions $\bar{n}_c$ that need to be explored. Given that the average time to compute $\Lambda_{\bar{n}_c}(D)$ for a given $\bar{n}_c$ is $10\mu s$, and that the number of different $\bar{n}_c$ (excluding $\bar{n}_c$ which are permutation of each other) for $n_c = 6$ is larger than 10^{11} for both Scenario B and Scenario C, it would roughly take 280 hours to obtain the results. In contrast, for the same example, the CC method explores less than 10^3 different $\bar{n}_c$ (see Section 8.5), and thus produces the results in less than $10ms$. Given that the CC method finds, in most cases, the optimal $\bar{n}_c^*$ in substantially shorter time shows that it is a good heuristic.

Comparing the results from the CC method with the results from the LEQC method shows that the CC method is always able to find a distribution $\bar{n}_c$ which results in an LoC that is higher or at least equal to the LoC obtained from the LEQC method (observe Tables 8.7–8.9). Important to note is that for n_c values close to the upper bound, both EQC and CC achieve the same LoC, although $\bar{n}_c$ is not same (observe $n_c \in [24, 25]$ for Table 8.7, $n_c \in [29, 30]$ for Table 8.8 and $n_c \in [14, 15]$ for Table 8.9). For such n_c values, no re-execution of execution segments of any size is possible without violating the deadline, and therefore all $\bar{n}_c$ produce the same LoC.

The major observation for all scenarios given in Table 8.3 is that the maximal LoC is achieved when the checkpoints are not evenly distributed. For example, in Table 8.7 the maximal LoC is achieved for $n_c = 13$ while using $\tilde{n}_c = [6^7, 9^2, 10^4]$, *i.e.* seven execution segments of size 6 t.u., two execution segments of size 9 t.u., and four execution segments of size 10 t.u.

The results indicate that non-equidistant checkpointing can improve the LoC when compared against LEQC. An important implication is that by using non-equidistant checkpointing the LoC can be improved in addition to achieving a shorter best case execution time, *i.e.* completion time when no errors occur $T + n_c \times \tau$. For example, in Table 8.8, the maximal LoC for LEQC is achieved for $n_c = 25$, but higher LoC can be achieved for $n_c \in [7, 24]$ when non-equidistant checkpointing (the CC method) is used (the highest LoC is achieved for $n_c = 15$ with the CC method). The same observation follows from Table 8.7 and Table 8.9.

n_c	LEQC	Non-Equidistant Checkpointing	
		EXS	CC
2	1.99999e-5 $\tilde{n}_c = [50^2]$	**1.12000e-5** $\tilde{n}_c = [44^1, 56^1]$	**1.12000e-5** $\tilde{n}_c = [44^1, 56^1]$
3	2.66678e-10 $\tilde{n}_c = [33^2, 34^1]$	**2.55998e-10** $\tilde{n}_c = [20^1, 40^2]$	**2.55998e-10** $\tilde{n}_c = [20^1, 40^2]$
4	2.49998e-10 $\tilde{n}_c = [25^4]$	**2.09559e-10** $\tilde{n}_c = [19^2, 31^2]$	**2.09559e-10** $\tilde{n}_c = [19^2, 31^2]$
5	2.39998e-10 $\tilde{n}_c = [20^5]$	**1.12000e-10** $\tilde{n}_c = [18^4, 28^1]$	**1.12000e-10** $\tilde{n}_c = [18^4, 28^1]$
6	2.07477e-15 $\tilde{n}_c = [16^2, 17^4]$	**2.07477e-15** $\tilde{n}_c = [16^2, 17^4]$	**2.07477e-15** $\tilde{n}_c = [16^2, 17^4]$
7	1.95991e-15 $\tilde{n}_c = [14^5, 15^2]$	**1.95161e-15** $\tilde{n}_c = [7^1, 15^3, 16^3]$	**1.95161e-15** $\tilde{n}_c = [7^1, 15^3, 16^3]$
8	1.87599e-15 $\tilde{n}_c = [12^4, 13^4]$	**1.83839e-15** $\tilde{n}_c = [7^2, 14^4, 15^2]$	**1.83839e-15** $\tilde{n}_c = [7^2, 14^4, 15^2]$
9	1.81113e-15 $\tilde{n}_c = [11^8, 12^1]$	**1.70906e-15** $\tilde{n}_c = [6^3, 13^2, 14^4]$	**1.70906e-15** $\tilde{n}_c = [6^3, 13^2, 14^4]$
10	1.75999e-15 $\tilde{n}_c = [10^{10}]$	**1.55958e-15** $\tilde{n}_c = [8^6, 13^4]$	**1.55958e-15** $\tilde{n}_c = [8^6, 13^4]$
11	1.71943e-15 $\tilde{n}_c = [9^{10}, 10^1]$	**1.58463e-15** $\tilde{n}_c = [5^4, 8^1, 12^6]$	1.58908e-15 $\tilde{n}_c = [5^4, 11^4, 12^3]$
12	1.68642e-15 $\tilde{n}_c = [8^8, 9^4]$	**1.53599e-15** $\tilde{n}_c = [5^4, 10^8]$	**1.53599e-15** $\tilde{n}_c = [5^4, 10^8]$
13	1.65807e-15 $\tilde{n}_c = [7^4, 8^9]$	**1.53236e-15** $\tilde{n}_c = [6^7, 9^2, 10^4]$	**1.53236e-15** $\tilde{n}_c = [6^7, 9^2, 10^4]$
14	1.63343e-15 $\tilde{n}_c = [7^{12}, 8^2]$	**1.57695e-15** $\tilde{n}_c = [4^3, 8^{11}]$	**1.57695e-15** $\tilde{n}_c = [4^3, 8^{11}]$
15	1.61335e-15 $\tilde{n}_c = [6^5, 7^{10}]$	**1.57695e-15** $\tilde{n}_c = [3^4, 8^{11}]$	**1.57695e-15** $\tilde{n}_c = [3^4, 8^{11}]$
16	**1.59510e-15** $\tilde{n}_c = [6^{12}, 7^4]$	**1.59510e-15** $\tilde{n}_c = [6^{12}, 7^4]$	**1.59510e-15** $\tilde{n}_c = [6^{12}, 7^4]$
17	**1.57863e-15** $\tilde{n}_c = [5^2, 6^{15}]$	**1.57863e-15** $\tilde{n}_c = [5^2, 6^{15}]$	**1.57863e-15** $\tilde{n}_c = [5^2, 6^{15}]$
18	1.75199e-10 $\tilde{n}_c = [5^8, 6^{10}]$	**7.60007e-11** $\tilde{n}_c = [5^{16}, 10^2]$	**7.60007e-11** $\tilde{n}_c = [5^{16}, 10^2]$
19	2.10599e-10 $\tilde{n}_c = [5^{14}, 6^5]$	**1.36000e-10** $\tilde{n}_c = [4^{15}, 10^4]$	**1.36000e-10** $\tilde{n}_c = [4^{15}, 10^4]$

to be continued on next page

n_c	LEQC	Non-Equidistant Checkpointing	
		EXS	CC
20	2.09999e-10 $\tilde{n}_c = [5^{20}]$	**1.84319e-10** $\tilde{n}_c = [3^{12}, 8^8]$	**1.84319e-10** $\tilde{n}_c = [3^{12}, 8^8]$
21	2.09599e-10 $\tilde{n}_c = [4^5, 5^{16}]$	**2.07479e-10** $\tilde{n}_c = [2^6, 5^2, 6^{13}]$	**2.07479e-10** $\tilde{n}_c = [2^6, 5^2, 6^{13}]$
22	1.20000e-5 $\tilde{n}_c = [4^{10}, 5^{12}]$	**3.20015e-6** $\tilde{n}_c = [4^{21}, 16^1]$	**3.20015e-6** $\tilde{n}_c = [4^{21}, 16^1]$
23	1.99999e-5 $\tilde{n}_c = [4^{15}, 5^8]$	**1.12000e-5** $\tilde{n}_c = [2^{22}, 56^1]$	**1.12000e-5** $\tilde{n}_c = [2^{22}, 56^1]$
24	**1.99999e-5** $\tilde{n}_c = [4^{20}, 5^4]$	**1.99999e-5** $\tilde{n}_c = [4^{20}, 5^4]$	**1.99999e-5** $\tilde{n}_c = [4^{23}, 8^1]$
25	**1.99999e-5** $\tilde{n}_c = [4^{25}]$	**1.99999e-5** $\tilde{n}_c = [4^{25}]$	**1.99999e-5** $\tilde{n}_c = [4^{25}]$

Table 8.7.: Comparison of $\bar{\Lambda}_{n_c}(D)$ for different checkpointing schemes at different n_c values for Scenario A

n_c	LEQC	Non-Equidistant Checkpointing	
		EXS	CC
2	1.99999e-5 $\tilde{n}_c = [500^2]$	**1.45999e-5** $\tilde{n}_c = [270^1, 730^1]$	**1.45999e-5** $\tilde{n}_c = [270^1, 730^1]$
3	1.99999e-5 $\tilde{n}_c = [333^2, 334^1]$	**9.60008e-6** $\tilde{n}_c = [260^2, 480^1]$	**9.60008e-6** $\tilde{n}_c = [260^2, 480^1]$
4	**2.49998e-10** $\tilde{n}_c = [250^4]$	**2.49998e-10** $\tilde{n}_c = [250^4]$	**2.49998e-10** $\tilde{n}_c = [250^4]$
5	2.39998e-10 $\tilde{n}_c = [200^5]$	**2.36515e-10** $\tilde{n}_c = [115^1, 221^3, 222^1]$	**2.36515e-10** $\tilde{n}_c = [115^1, 221^3, 222^1]$
6	2.33332e-10 $\tilde{n}_c = [166^2, 167^4]$	$\emptyset$	**2.08146e-10** $\tilde{n}_c = [110^3, 223^2, 224^1]$
7	2.28570e-10 $\tilde{n}_c = [142^1, 143^6]$	$\emptyset$	**1.87146e-10** $\tilde{n}_c = [105^4, 193^2, 194^1]$
8	2.24999e-10 $\tilde{n}_c = [125^8]$	$\emptyset$	**1.44000e-10** $\tilde{n}_c = [100^6, 200^2]$
9	2.22221e-10 $\tilde{n}_c = [111^8, 112^1]$	$\emptyset$	**1.22777e-10** $\tilde{n}_c = [95^7, 167^1, 168^1]$
10	2.19999e-10 $\tilde{n}_c = [100^{10}]$	$\emptyset$	**7.60008e-11** $\tilde{n}_c = [90^9, 190^1]$
11	2.18181e-10 $\tilde{n}_c = [90^1, 91^{10}]$	$\emptyset$	**6.00009e-11** $\tilde{n}_c = [85^{10}, 150^1]$
12	2.16666e-10 $\tilde{n}_c = [83^8, 84^4]$	$\emptyset$	**4.80011e-11** $\tilde{n}_c = [80^{11}, 120^1]$
13	2.15383e-10 $\tilde{n}_c = [76^1, 77^{12}]$	$\emptyset$	**4.00011e-11** $\tilde{n}_c = [75^{12}, 100^1]$
14	2.14285e-10 $\tilde{n}_c = [71^8, 72^6]$	$\emptyset$	**3.60012e-11** $\tilde{n}_c = [70^{13}, 90^1]$
15	2.13333e-10 $\tilde{n}_c = [66^5, 67^{10}]$	$\emptyset$	**3.60011e-11** $\tilde{n}_c = [65^{14}, 90^1]$
16	2.12500e-10 $\tilde{n}_c = [62^8, 63^8]$	$\emptyset$	**4.00011e-11** $\tilde{n}_c = [60^{15}, 100^1]$
17	2.11764e-10 $\tilde{n}_c = [58^3, 59^{14}]$	$\emptyset$	**4.80010e-11** $\tilde{n}_c = [55^{16}, 120^1]$
18	2.11111e-10 $\tilde{n}_c = [55^8, 56^{10}]$	$\emptyset$	**7.60007e-11** $\tilde{n}_c = [50^{16}, 100^2]$
19	2.10526e-10 $\tilde{n}_c = [52^7, 53^{12}]$	$\emptyset$	**1.01547e-10** $\tilde{n}_c = [45^{16}, 93^2, 94^1]$

to be continued on next page

n_c	LEQC	Non-Equidistant Checkpointing	
		EXS	CC
20	2.09999e-10 $\tilde{n}_c = [50^{20}]$	$\emptyset$	**1.24560e-10** $\tilde{n}_c = [40^{16}, 90^4]$
21	2.09524e-10 $\tilde{n}_c = [47^8, 48^{13}]$	$\emptyset$	**1.52395e-10** $\tilde{n}_c = [35^{15}, 79^5, 80^1]$
22	2.09091e-10 $\tilde{n}_c = [45^{12}, 46^{10}]$	$\emptyset$	**1.77848e-10** $\tilde{n}_c = [30^{13}, 67^2, 68^7]$
23	2.08696e-10 $\tilde{n}_c = [43^{12}, 44^{11}]$	$\emptyset$	**1.96153e-10** $\tilde{n}_c = [25^{10}, 57^4, 58^9]$
24	2.08333e-10 $\tilde{n}_c = [41^8, 42^{16}]$	$\emptyset$	**2.05724e-10** $\tilde{n}_c = [20^6, 48^2, 49^{16}]$
25	**2.07999e-10** $\tilde{n}_c = [40^{25}]$	$\emptyset$	**2.07999e-10** $\tilde{n}_c = [40^{25}]$
26	1.99999e-5 $\tilde{n}_c = [38^{14}, 39^{12}]$	$\emptyset$	**5.00012e-6** $\tilde{n}_c = [30^{25}, 250^1]$
27	1.99999e-5 $\tilde{n}_c = [37^{26}, 38^1]$	$\emptyset$	**9.60005e-6** $\tilde{n}_c = [20^{26}, 480^1]$
28	1.99999e-5 $\tilde{n}_c = [35^8, 36^{20}]$	$\emptyset$	**1.45999e-5** $\tilde{n}_c = [10^{27}, 730^1]$
29	**1.99999e-5** $\tilde{n}_c = [34^{15}, 35^{14}]$	$\emptyset$	**1.99999e-5** $\tilde{n}_c = [34^{28}, 48^1]$
30	**1.99999e-5** $\tilde{n}_c = [33^{20}, 34^{10}]$	$\emptyset$	**1.99999e-5** $\tilde{n}_c = [33^{29}, 43^1]$

Table 8.8.: Comparison of $\bar{\Lambda}_{n_c}(D)$ for different checkpointing schemes at different n_c values for Scenario B

n_c	LEQC	Non-Equidistant Checkpointing	
		EXS	CC
2	1.99999e-5 $\tilde{n}_c = [500^2]$	**1.51999e-5** $\tilde{n}_c = [240^1, 760^1]$	**1.51999e-5** $\tilde{n}_c = [240^1, 760^1]$
3	1.99999e-5 $\tilde{n}_c = [333^2, 334^1]$	**1.12000e-5** $\tilde{n}_c = [220^2, 560^1]$	**1.12000e-5** $\tilde{n}_c = [220^2, 560^1]$
4	1.99999e-5 $\tilde{n}_c = [250^4]$	**8.00010e-6** $\tilde{n}_c = [200^3, 400^1]$	**8.00010e-6** $\tilde{n}_c = [200^3, 400^1]$
5	1.99999e-5 $\tilde{n}_c = [200^5]$	**5.60014e-6** $\tilde{n}_c = [180^4, 280^1]$	**5.60014e-6** $\tilde{n}_c = [180^4, 280^1]$
6	1.99999e-5 $\tilde{n}_c = [166^2, 167^4]$	$\emptyset$	**4.00016e-6** $\tilde{n}_c = [160^5, 200^1]$
7	1.99999e-5 $\tilde{n}_c = [142^1, 143^6]$	$\emptyset$	**3.20017e-6** $\tilde{n}_c = [140^6, 160^1]$
8	1.99999e-5 $\tilde{n}_c = [125^8]$	$\emptyset$	**3.20017e-6** $\tilde{n}_c = [120^7, 160^1]$
9	1.99999e-5 $\tilde{n}_c = [111^8, 112^1]$	$\emptyset$	**4.00015e-6** $\tilde{n}_c = [100^8, 200^1]$
10	1.99999e-5 $\tilde{n}_c = [100^{10}]$	$\emptyset$	**5.60012e-6** $\tilde{n}_c = [80^9, 280^1]$
11	1.99999e-5 $\tilde{n}_c = [90^1, 91^{10}]$	$\emptyset$	**8.00008e-6** $\tilde{n}_c = [60^{10}, 400^1]$
12	1.99999e-5 $\tilde{n}_c = [83^8, 84^4]$	$\emptyset$	**1.12000e-5** $\tilde{n}_c = [40^{11}, 560^1]$
13	1.99999e-5 $\tilde{n}_c = [76^1, 77^{12}]$	$\emptyset$	**1.51999e-5** $\tilde{n}_c = [20^{12}, 760^1]$
14	**1.99999e-5** $\tilde{n}_c = [71^8, 72^6]$	$\emptyset$	**1.99999e-5** $\tilde{n}_c = [71^{13}, 77^1]$
15	**1.99999e-5** $\tilde{n}_c = [66^5, 67^{10}]$	$\emptyset$	**1.99999e-5** $\tilde{n}_c = [66^{14}, 76^1]$

Table 8.9.: Comparison of $\bar{\Lambda}_{n_c}(D)$ for different checkpointing schemes at different n_c values for Scenario C

9

Summary of Part II

In Part II, we discussed optimization of RRC for two different optimization objectives while assuming that the checkpoints are not evenly distributed throughout the execution of a job, *i.e.* non-equidistant checkpointing. The optimization objectives discussed in Part II are: (1) *Average Execution Time* and (2) *Level of Confidence*. Next, we outline the contributions with respect to each of the optimization objectives stated earlier.

For the first optimization objective, *i.e. Average Execution Time* (AET), we proposed two techniques that estimate the error probability over time and based on the estimates adjust the checkpointing scheme with the goal to reduce the AET. The two proposed techniques are: Periodic Probability Estimation (PPE) and Aperiodic Probability Estimation (APE). The consequence of using these techniques is that since the checkpointing scheme is adjusted over time, the checkpoints will not be evenly distributed throughout the execution of the job. This implies usage of non-equidistant checkpointing. We conducted experiments and showed that both techniques can make a good estimation on the error probability and thus significantly reduce the AET. The APE approach provided slightly better results (lower AET) when compared against the PPE approach.

For the second optimization objective, *i.e. Level of Confidence* (LoC), we analyzed the impact on the probability of meeting a given deadline, *i.e.* the LoC, when the checkpoints are not evenly distributed. First, we developed a mathematical expression to calculate the LoC for a given distribution of a number of checkpoints (a vector where each element defines the size of each execution segment). Since the number of possible distributions increases drastically, running exhaustive search to obtain the optimal distribution which results in the maximal LoC is not always possible. Therefore, we proposed a method, *i.e.* Clustered Checkpointing, that maximizes the LoC at much lower computation cost when compared against the exhaustive search method. Experiments were

conducted and results show that the proposed method is able in most cases to find the optimal distribution of the checkpoints that results in the maximal LoC.

Part III

Thesis Summary

10

Conclusions and Future Work

Increasing soft error rates observed in the latest semiconductor technologies enforce the usage of fault tolerance in various computer systems. While fault tolerance enables correct operation even in the presence of errors, the major drawback is that it often introduces a time overhead. The introduced time overhead impacts negatively a particular class of computer systems commonly referred to as Real-Time Systems (RTSs), where correct operation is defined as producing the correct outcome while ensuring time constraints. Therefore, for RTSs it is important to optimize the usage of fault tolerance such that the negative effect of the introduced time overhead is minimized.

Fault tolerance is a well established research topic and it offers a wide variety of techniques to achieve correct operation even in the presence of errors. Roll-back Recovery with Checkpointing (RRC) is one technique that efficiently copes with soft errors. The advantage of this technique over other fault tolerance techniques is that (1) it is not as costly as other techniques which require a significant amount of hardware redundancy, and (2) in case of errors, RRC only re-executes a portion of the job that is being executed instead of restarting the job from the beginning, which is usually done in other techniques. The main drawback of RRC is that it introduces a time overhead which depends on the number of checkpoints that are used.

In this thesis, we studied the usage of RRC in RTSs. The goal of the thesis is to provide an optimization framework where RRC can be optimized based on different optimization objectives. Since in RRC the checkpoints can either be distributed evenly or unevenly over the execution of the job, we considered the two schemes: equidistant and non-equidistant checkpointing. The presented optimization framework considers optimization of RRC when equidistant and non-equidistant checkpointing is used. The following optimization objectives are considered: (1) *Average Execution Time*, (2) *Level of Confidence*, and (3) *Guaranteed Completion Time*.

The purpose of the presented optimization framework is to assist a designer of an RTS, in the early design stage, with the following. First, to help the designer decide if RRC is a suitable choice with respect to the different specification requirements that the designer needs to deal with. Second, if RRC is suitable, then this optimization framework should give the knowledge to the designer of an RTS on how to select the number of checkpoints to be used and how these checkpoints should be distributed, such that the specification requirements are met.

The rest of the chapter is organized as follows. First, we provide conclusions with respect to each of the different optimization objectives considered in this thesis. Finally, at the end of this chapter, we outline directions for future work.

10.1. AVERAGE EXECUTION TIME

Average Execution Time (AET) is an optimization objective which is mostly suitable for soft RTSs where eventual deadline violation does not result in severe consequences. With respect to AET, we showed that for an equidistant checkpointing scheme it is possible to identify the optimal number of checkpoints such that the minimal AET is achieved. Mathematical expressions were derived to calculate the optimal number of checkpoints and the minimal AET. As presented, both expressions depend on multiple parameters such as checkpointing overhead, processing time and the probability that no errors occur in a processing node within an interval of time. While the checkpointing overhead and the processing time can be considered invariable over time, the probability of errors may vary over time. In such scenario, non-equidistant checkpointing is more preferable instead of equidistant checkpointing. Using adaptive techniques, such as the methods that we proposed in Chapter 7, which estimate the error probability over time and correspondingly adjust the checkpointing scheme, can provide better results in terms of reducing the AET. Observe that even though non-equidistant checkpointing is preferred over equidistant checkpointing with respect to AET, this does not diminish the results obtained from optimizing RRC with respect to AET for the equidistant checkpointing scheme. The adaptive techniques can estimate the error probability over time and then apply equidistant checkpointing with the optimal number of checkpoints computed for the latest estimate of the error probability until a new estimate of the error probability is provided. Both methods presented in Chapter 7, *i.e.* Periodic Probability Estimation (PPE) and Aperiodic Probability Estimation (APE), used such a scheme and as it was shown in the experimental results section, these methods resulted in significant reduction of the AET where the APE method performed slightly better than the PPE method.

10.2. LEVEL OF CONFIDENCE

Level of Confidence (LoC) is a useful metric to measure to what extent deadlines in RTSs are met. In contrast to AET, which is mostly suitable for soft RTSs, LoC is equally applicable for both soft and hard RTSs. For soft RTSs, the highest LoC may not be needed. However, for hard RTSs it is very important that the LoC is almost equal to one. Therefore, to achieve the maximal LoC, we considered optimization of RRC with the goal to maximize the LoC.

With respect to LoC, the important conclusions are as follows. The LoC varies with the number of checkpoints used, and there exists an optimal number of checkpoints that results in the maximal LoC. We studied, for the case of equidistant checkpointing, the impact of the number of checkpoints on the LoC for two scenarios, *i.e.* (1) a single job with a deadline and (2) multiple jobs (a set of jobs) with a global deadline. As shown in Chapter 4, the conclusion is that optimizing RRC for a single job does not solve the problem of multiple jobs. Two approaches that aim to reduce the problem of multiple jobs to a single job were investigated and it was shown that neither of them is able to optimally solve the problem of multiple jobs. In the first approach, the notion of local deadlines was introduced. A local deadline was assigned to each job in the set and the number of checkpoints for each job was selected such that the LoC with respect to the local deadline was maximized. The results showed that by using this approach there is no guarantee that such assignment of the number of checkpoints to the jobs will result in the maximal LoC with respect to the global deadline. In the second approach, the set of jobs was considered as a single large job. Using the optimization method for a single job, the optimal number of checkpoints that maximizes the LoC with respect to the global deadline was identified and these checkpoints were evenly distributed among the jobs in the set. The results showed that this method does not obtain, in the general case, the optimal assignment of checkpoints that results in the maximal LoC with respect to the global deadline. To solve the problem for the case of multiple jobs, we proposed a method to find the number of checkpoints for each job such that the LoC with respect to the global deadline is maximized. The important conclusion was that even though we assumed that all jobs in the set have equal processing time, the maximal LoC with respect to the global deadline was not achieved when the same number of checkpoints were assigned to the jobs. Instead, different number of checkpoints was assigned to different jobs. This implies that distributing the checkpoints evenly may not result in the maximal LoC. Therefore, in Chapter 8, we studied the impact on the LoC when the checkpoints are not evenly distributed, *i.e.* non-equidistant checkpointing. The analysis was done only for the scenario of a single job with a deadline. The results showed that non-equidistant checkpointing can always improve the LoC in comparison to equidistant checkpointing. Impor-

tant to point out is that the previous conclusion does not discard our findings on optimizing RRC with respect to LoC when equidistant checkpointing is used. Instead, we strongly advise that these two studies should be well combined. For example, optimizing RRC with respect to equidistant checkpoints makes it possible the find the optimal number of checkpoints that results in the highest LoC. Once this number of checkpoints is obtained, further improvement on the LoC can be achieved by distributing these checkpoints in a non-equidistant manner.

Another important conclusion that was drawn after studying the optimization of RRC with respect to LoC is the following. The optimal number of checkpoints that minimizes the AET decreases the LoC with respect to a given deadline, and the optimal number of checkpoints that maximizes the LoC with respect to a given deadline increases the AET.

10.3. GUARANTEED COMPLETION TIME

For soft RTSs, most of the work focuses on optimization of RRC with respect to AET. While there are advantages to consider the AET as an optimization objective, the main disadvantage with the AET is that it does not provide any information on what is the probability to meet a given deadline. However, a designer of a RTS may have some reliability constraints which dictate that deadlines must be met with some probability. In such case, optimization based on AET is not sufficient. Instead, it is more important to minimize the completion time while ensuring that some reliability constraints are met. For that reason, in Chapter 5, we introduced the concept Guaranteed Completion Time (GCT_δ) which refers to a completion time such that the probability that a job completes within this time is higher or equal to a given LoC requirement δ. In Chapter 5, we analyzed the GCT_δ and showed that GCT_δ varies with the number of checkpoints that are used. Furthermore, we showed that there exists an optimal number of checkpoints that minimizes the GCT_δ. For that reason, we developed a method to find the optimal number of checkpoints that results in the minimal GCT_δ. The advantage of the minimal GCT_δ over the minimal AET is that the minimal GCT_δ satisfies a given reliability constraint (a job will complete within an period of time equal to GCT_δ with a probability δ), while the minimal AET does not provide any information about the probability that a job will complete within a period of time equal to the minimal AET.

10.4. FUTURE WORK

While in this thesis the overall goal was to provide an optimization framework for RRC when used in RTSs, the presented optimization framework can be further improved. Therefore, in this section, we provide directions for future work in order to improve the optimization framework that was presented.

In this thesis, for both equidistant and non-equidistant checkpointing, we presented a mathematical model to evaluate the LoC with respect to a given deadline for a single job, and based on this model we proposed methods to obtain the optimal number of checkpoints that results in the maximal LoC. In the derived mathematical model, we assumed that the error-free probability over a period of time is fixed. However, the error-free probability may change over time (as discussed in Chapter 7), and this opens the following research questions. What is the impact on the LoC when the error-free probability changes over time? If the number of checkpoints used is obtained while using an estimate of the error-free probability, how should the checkpointing scheme (the number of checkpoints) change in order to ensure with sufficiently high LoC that deadlines will be met? These and similar questions provide a foundation for future research work.

Another important aspect to address is how to evaluate the LoC with respect to a given deadline for multiple jobs. In this thesis, only for equidistant checkpointing we have addressed evaluation of LoC with respect to a given deadline for multiple jobs, but we have not addressed this problem for non-equidistant checkpointing. Furthermore, for the multiple job case that is addressed in this thesis, we have considered a set of jobs that are executed in a sequence. However, exploiting the multi-processor paradigm, it may be possible to execute non-dependent jobs on different processors concurrently. Therefore, another interesting problem to consider is how to evaluate the LoC with respect to a given deadline for a set of jobs, where some jobs are executed sequentially and some jobs are executed concurrently. How many checkpoints should be assigned to each job, and how these checkpoints should be distributed such that the LoC with respect to the given deadline is maximized is another optimization problem that can improve the optimization framework that is presented in this thesis.

Another research direction that has not been addressed in this thesis is how to minimize the GCT_δ when non-equidistant checkpointing is used. Thus, the framework can be improved by providing results (methods) to identify how many checkpoints to be used and how these checkpoints should be distributed in order to minimize the GCT_δ for a given LoC requirement δ.

So far, we discussed some research directions in order to broaden the scope of the optimization framework that was presented in this thesis. Next, we present a research direction in order to strengthen the value of the pre-

sented work. The optimization framework presented in this thesis has only been tested on synthetic benchmarks (mathematical models and simulations). However, to strengthen the value of this work it is important to test the presented optimization framework with practical benchmarks. Therefore, the next goal is to implement RRC and validate the results presented in this thesis on more realistic examples.

Part IV

Appendix

Appendix A

In this section we prove, that the following expression:

$$p_{n_c}(k) = N_{n_c}(k) \times P_\epsilon^{n_c} \times (1 - P_\epsilon)^k = \binom{n_c + k - 1}{k} \times P_\epsilon^{n_c} \times (1 - P_\epsilon)^k \quad \text{(A.1)}$$

can be used as a valid probability distribution function, due to the fact that it satisfies the following necessary condition which a probability distribution function must satisfy:

$$\sum_{k=0}^{\infty} p_{n_c}(k) = 1 \quad \text{(A.2)}$$

To ensure that the sum of all terms from the probability distribution function $p_{n_c}(k)$ is finite (Eq. (A.2)), we need to check if the terms of $p_{n_c}(k)$ converge. The property of convergence can be checked using the following expression:

$$\lim_{k \to \infty} \frac{p_{n_c}(k+1)}{p_{n_c}(k)} < 1 \quad \text{(A.3)}$$

Next, we show that this property (Eq. (A.3)) is satisfied:

$$\begin{aligned} & \lim_{k \to \infty} \frac{p_{n_c}(k+1)}{p_{n_c}(k)} < 1 \\ \Rightarrow \quad & \lim_{k \to \infty} \frac{\binom{n_c + k}{k+1} P_\epsilon^{n_c} (1 - P_\epsilon)^{k+1}}{\binom{n_c + k - 1}{k} P_\epsilon^{n_c} (1 - P_\epsilon)^k} \end{aligned}$$

$$
\begin{aligned}
&= \lim_{k\to\infty} \frac{\frac{(n_c+k)!}{(k+1)!(n_c-1)!} P_\epsilon^{n_c}(1-P_\epsilon)^{k+1}}{\frac{(n_c+k-1)!}{k!(n_c-1)!} P_\epsilon^{n_c}(1-P_\epsilon)^{k}} \\
&= \lim_{k\to\infty} \frac{n_c+k}{k+1}(1-P_\epsilon) \\
&= (1-P_\epsilon) \lim_{k\to\infty} \frac{k(1+\frac{n_c}{k})}{k(1+\frac{1}{k})} \\
&= (1-P_\epsilon) \frac{1+\lim\limits_{k\to\infty} \frac{n_c}{k}}{1+\lim\limits_{k\to\infty} \frac{1}{k}} \\
&= (1-P_\epsilon) < 1
\end{aligned} \tag{A.4}
$$

Since the convergence property is satisfied, we can continue with the proof to show that the sum of all terms of $p_{n_c}(k)$ must be equal to one, (Eq. (A.2)). To prove this we use mathematical induction. We introduce a new variable Q_ϵ, such that $Q_\epsilon = 1 - P_\epsilon$, and we assume that the following expression holds:

$$
\begin{aligned}
&\sum_{k=0}^{\infty} \binom{n_c+k-1}{k} Q_\epsilon^k (1-Q_\epsilon)^{n_c} = 1 \\
\equiv \\
&\sum_{k=0}^{\infty} \binom{n_c+k-1}{k} Q_\epsilon^k = \frac{1}{(1-Q_\epsilon)^{n_c}}
\end{aligned} \tag{A.5}
$$

Let n_c=1, then the following expression gives the left-hand side of Eq. (A.5):

$$
\begin{aligned}
&\sum_{k=0}^{\infty} \binom{1+k-1}{k} Q_\epsilon^k \\
\equiv\ &\sum_{k=0}^{\infty} \binom{k}{k} Q_\epsilon^k = \sum_{k=0}^{\infty} Q_\epsilon^k
\end{aligned} \tag{A.6}
$$

This expression represents a sum of geometric series, and this sum is finite and it is calculated as:

$$
\sum_{k=0}^{\infty} Q_\epsilon^k = \frac{1}{1-Q_\epsilon} \tag{A.7}
$$

The result in Eq. (A.7) corresponds to the right-hand side of the assumption that we presented earlier in Eq. (A.5) for $n_c = 1$.

Let $n_c = 2$, then the left-hand side of Eq. (A.5) is evaluated as:

$$\begin{aligned} & \sum_{k=0}^{\infty} \binom{k+1}{k} Q_\epsilon^k \\ \equiv \quad & \sum_{k=0}^{\infty} \frac{(k+1)!}{k!1!} Q_\epsilon^k = \sum_{k=0}^{\infty} (k+1) Q_\epsilon^k \\ \equiv \quad & \sum_{k=0}^{\infty} k Q_\epsilon^k + \sum_{k=0}^{\infty} Q_\epsilon^k \end{aligned} \tag{A.8}$$

By calculating the derivative of the sum presented in Eq. (A.7) we get:

$$\begin{aligned} \Rightarrow \quad & \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(\sum_{k=0}^{\infty} Q_\epsilon^k \right) = \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(\frac{1}{1-Q_\epsilon} \right) \\ \Rightarrow \quad & \sum_{k=0}^{\infty} \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} Q_\epsilon^k = \frac{1}{(1-Q_\epsilon)^2} \\ \Rightarrow \quad & \sum_{k=0}^{\infty} k Q_\epsilon^{k-1} = \frac{1}{(1-Q_\epsilon)^2} \\ \Rightarrow \quad & \frac{1}{Q_\epsilon} \sum_{k=0}^{\infty} k {Q_\epsilon}^k = \frac{1}{(1-Q_\epsilon)^2} \\ \Rightarrow \quad & \sum_{k=0}^{\infty} k {Q_\epsilon}^k = \frac{Q_\epsilon}{(1-Q_\epsilon)^2} \end{aligned} \tag{A.9}$$

So going back to Eq. (A.8), and by using Eq. (A.9) and Eq. (A.7) we get:

$$\begin{aligned} & \sum_{k=0}^{\infty} \binom{k+1}{k} Q_\epsilon^k \\ = \quad & \sum_{k=0}^{\infty} k Q_\epsilon^k + \sum_{k=0}^{\infty} Q_\epsilon^k \\ = \quad & \frac{Q_\epsilon}{(1-Q_\epsilon)^2} + \frac{1}{1-Q_\epsilon} \\ = \quad & \frac{Q_\epsilon + 1 - Q_\epsilon}{(1-Q_\epsilon)^2} \\ = \quad & \frac{1}{(1-Q_\epsilon)^2} \end{aligned} \tag{A.10}$$

The result in Eq. (A.10) adheres to the assumption that we presented earlier in Eq. (A.5) for $n_c = 2$ (observe the right-hand side of Eq. (A.5) for $n_c = 2$).

Next, we perform the inductive step by replacing n_c with $n_c + 1$, thus the left-hand side of Eq. (A.5) is evaluated as:

$$\begin{aligned}
&\sum_{k=0}^{\infty} \binom{n_c + k}{k} Q_\epsilon^k \\
= &\sum_{k=0}^{\infty} \frac{(n_c + k)!}{k! n_c!} Q_\epsilon^k \\
= &\sum_{k=0}^{\infty} \frac{(n_c + k)(n_c + k - 1)!}{n_c (n_c - 1)! k!} Q_\epsilon^k \\
= &\sum_{k=0}^{\infty} \frac{(n_c + k - 1)!}{(n_c - 1)! k!} Q_\epsilon^k + \sum_{k=0}^{\infty} \frac{k(n_c + k - 1)!}{n_c (n_c - 1)! k!} Q_\epsilon^k
\end{aligned} \tag{A.11}$$

The first term in Eq. (A.11) is the same as the assumption presented in Eq. (A.5). To evaluate the second term in Eq. (A.11) we need to calculate the derivative of the expression presented in Eq. (A.5). The derivative of the expression presented in Eq. (A.5) is given in Eq. (A.12).

$$\begin{aligned}
&\frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(\sum_{k=0}^{\infty} \frac{(n_c + k - 1)!}{(n_c - 1)! k!} Q_\epsilon^k \right) = \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(\frac{1}{(1 - Q_\epsilon)^{n_c}} \right) \\
\Rightarrow &\sum_{k=0}^{\infty} \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(\frac{(n_c + k - 1)!}{(n_c - 1)! k!} Q_\epsilon^k \right) = \frac{n_c}{(1 - Q_\epsilon)^{n_c + 1}} \\
\Rightarrow &\sum_{k=0}^{\infty} \frac{(n_c + k - 1)!}{(n_c - 1)! k!} \frac{\mathrm{d}}{\mathrm{d}Q_\epsilon} \left(Q_\epsilon^k \right) = \frac{n_c}{(1 - Q_\epsilon)^{n_c + 1}} \\
\Rightarrow &\sum_{k=0}^{\infty} \frac{(n_c + k - 1)!}{(n_c - 1)! k!} k Q_\epsilon^{k-1} = \frac{n_c}{(1 - Q_\epsilon)^{n_c + 1}} \\
\Rightarrow &\frac{1}{Q_\epsilon} \sum_{k=0}^{\infty} \frac{k(n_c + k - 1)!}{(n_c - 1)! k!} Q_\epsilon^k = \frac{n_c}{(1 - Q_\epsilon)^{n_c + 1}} \\
\Rightarrow &\sum_{k=0}^{\infty} \frac{k(n_c + k - 1)!}{(n_c - 1)! k!} Q_\epsilon^k = \frac{n_c Q_\epsilon}{(1 - Q_\epsilon)^{n_c + 1}}
\end{aligned} \tag{A.12}$$

By replacing the terms in Eq. (A.11) with Eq. (A.5) and Eq. (A.12), we get:

$$\begin{aligned}
&\sum_{k=0}^{\infty} \binom{n_c + k}{k} Q_\epsilon^k \\
= &\sum_{k=0}^{\infty} \frac{(n_c + k - 1)!}{(n_c - 1)! k!} Q_\epsilon^k + \sum_{k=0}^{\infty} \frac{k(n_c + k - 1)!}{n_c (n_c - 1)! k!} Q_\epsilon^k
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{k=0}^{\infty} \frac{(n_c+k-1)!}{(n_c-1)!k!} Q_\epsilon^k + \frac{1}{n_c} \sum_{k=0}^{\infty} \frac{k(n_c+k-1)!}{(n_c-1)!k!} Q_\epsilon^k \\
&= \frac{1}{(1-Q_\epsilon)^{n_c}} + \frac{1}{n_c} \frac{n_c Q_\epsilon}{(1-Q_\epsilon)^{n_c+1}} \\
&= \frac{(1-Q_\epsilon)+Q_\epsilon}{(1-Q_\epsilon)^{n_c+1}} \\
&= \frac{1}{(1-Q_\epsilon)^{n_c+1}} \qquad \text{(A.13)}
\end{aligned}
$$

By this (the expression obtained in Eq. (A.13) is the same as the right-hand side of the expression presented in Eq. (A.5) when using $n_c + 1$) we prove that the proposed function, given in (Eq. (A.1)), satisfies the necessary condition presented in Eq. (A.2), and therefore can be used as a probability distribution function.

Appendix B

In this section, we provide the proof for the following theorem:

Theorem 3. *For any k, n_{c_1}, and n_{c_2}, such that $n_{c_1} > n_{c_2}$, the following condition holds:*

$$\lambda_{n_{c_1}}(k) > \lambda_{n_{c_2}}(k)$$

Proof. **Theorem 3** states that for a fixed number of re-executions k, the LoC $\lambda_{n_c}(k)$ increases along with the number of checkpoints n_c.

The proof is based on the following lemmas:

Lemma 1. *For any $n_{c_1} > n_{c_2}$, the following condition holds: $p_{n_{c_1}}(1) > p_{n_{c_2}}(1)$.*

Lemma 2. *For any $0 < P_T < 1$, there exists a K_l, such that $P_T < e^{-\frac{K_l - 1}{2}}$, and for any $n_{c_1} > n_{c_2}$ and $k \leq K_l$ the following condition holds $p_{n_{c_1}}(k) > p_{n_{c_2}}(k)$.*

Lemma 3. *For any $0 < P_T < 1$, there exists a K_u, such that $P_T > \sqrt[K_u]{K_u + 1} - 1$, and for any $n_{c_1} > n_{c_2}$ and $k \geq K_u$ the following condition holds $p_{n_{c_1}}(k) < p_{n_{c_2}}(k)$.*

Next, we provide the proof for each of the lemmas presented earlier, *i.e.* **Lemma 1**, **2** and **3**.

Lemma 1 states that the probability for a job to complete at time ${}^{n_c}t_1$, *i.e.* after a single re-execution, increases as the number of checkpoints n_c increases. The probability that a job completes after a single re-execution, assuming that n_c checkpoints are used, is calculated with the following expression:

$$p_{n_c}(1) = \binom{n_c}{1} \times {P_\epsilon}^{n_c} \times (1 - P_\epsilon) = n_c \times {P_T}^2 \times \left(1 - {P_T}^{\frac{2}{n_c}}\right) \tag{B.1}$$

To prove that $p_{n_c}(1)$ increases with n_c, we can calculate the first derivative of Eq. (B.1) with respect to n_c and show that it is greater than zero for any $n_c \geq 1$.

The first derivative of Eq. (B.1) is given with the following equation:

$$P_T{}^2 \left[1 - P_T{}^{\frac{2}{n_c}} + \ln P_T \times P_T{}^{\frac{2}{n_c}} \times \frac{2}{n_c}\right] \tag{B.2}$$

From Eq. (B.2), it is difficult to come to the conclusion that the expression in Eq. (B.2) is greater than zero for any $n_c \geq 1$. For that reason, we examine whether the expression in Eq. (B.2) is increasing or decreasing with n_c, by computing its first derivative and check if that expression is positive (greater than zero) or negative (less than zero) for all $n_c > 1$. The first derivative of Eq. (B.2) is evaluated as:

$$-P_T{}^{\frac{2+2n_c}{n_c}} \times \frac{4}{{n_c}^3} \times (\ln P_T)^2 \tag{B.3}$$

The expression in Eq. (B.3) is negative for all $n_c \geq 1$ due to the fact that each of the three multiplicands is evaluated as positive ($\frac{4}{{n_c}^3} > 0$ when $n_c \geq 1$, $(\ln P_T)^2 > 0$, and $P_T{}^{\frac{2+2n_c}{n_c}} > 0$). As the first derivative of Eq. (B.2) is negative (less than zero) for all $n_c \geq 1$, it means that the expression in Eq. (B.2) is monotonically decreasing with n_c.

Next, we examine the limit of Eq. (B.2) when n_c approaches infinity:

$$\begin{aligned}
\lim_{n_c \to \infty} \quad & P_T{}^2 \left[1 - P_T{}^{\frac{2}{n_c}} + \ln P_T \times P_T{}^{\frac{2}{n_c}} \times \frac{2}{n_c}\right] \\
= \quad & P_T{}^2 \left[1 - \lim_{n_c \to \infty} P_T{}^{\frac{2}{n_c}} + \ln P_T \times \lim_{n_c \to \infty} P_T{}^{\frac{2}{n_c}} \times \frac{2}{n_c}\right] \\
= \quad & P_T{}^2 \left[1 - 1 + \ln P_T \times 0\right] \\
= \quad & 0
\end{aligned} \tag{B.4}$$

So far, we have shown that the first derivative of Eq. (B.1) is monotonically decreasing for $n_c \geq 1$, and that it approaches zero as n_c approaches infinity. From this we conclude that the first derivative of Eq. (B.1) is greater than zero for all $n_c \geq 1$, which implies that the expression in Eq. (B.1) is increasing with n_c.

Next, we provide the proof for **Lemma 2**, which states that $p_{n_{c_1}}(k) > p_{n_{c_2}}(k)$ for any $n_{c_1} > n_{c_2}$ and any $k < K_l$, such that $P_T < e^{-\frac{K_l - 1}{2}}$. The proof is derived based on the ratio:

$$\frac{p_{2n_c}(k)}{p_{n_c}(k)} \tag{B.5}$$

The ratio in Eq. (B.5) is evaluated as:

$$
\begin{aligned}
\frac{p_{2n_c}(k)}{p_{n_c}(k)} &= \frac{\binom{2n_c+k-1}{k} \times {P_\epsilon}^{2n_c} \times (1-P_\epsilon)^k}{\binom{n_c+k-1}{k} \times {P_\epsilon}^{n_c} \times (1-P_\epsilon)^k} \\
&= \frac{\frac{\prod_{i=0}^{k-1}(2n_c+i)}{k!} \times \left({P_T}^{\frac{2}{2n_c}}\right)^{2n_c} \times \left(1-{P_T}^{\frac{2}{2n_c}}\right)^k}{\frac{\prod_{i=0}^{k-1}(n_c+i)}{k!} \times \left({P_T}^{\frac{2}{n_c}}\right)^{n_c} \times \left(1-{P_T}^{\frac{2}{n_c}}\right)^k} \\
&= \frac{\prod_{i=0}^{k-1}(2n_c+i) \times {P_T}^2 \times \left(1-{P_T}^{\frac{1}{n_c}}\right)^k}{\prod_{i=0}^{k-1}(n_c+i) \times {P_T}^2 \times \left(1-{P_T}^{\frac{1}{n_c}}\right)^k \times \left(1+{P_T}^{\frac{1}{n_c}}\right)^k} \\
&= \prod_{i=0}^{k-1}\left(\frac{2n_c+i}{n_c+i}\right) \times \frac{1}{\left(1+{P_T}^{\frac{1}{n_c}}\right)^k} \qquad \text{(B.6)}
\end{aligned}
$$

When the ratio in Eq. (B.5) is greater than one implies that $p_{2n_c}(k) > p_{n_c}(k)$. Hence, we get the following inequality:

$$
\prod_{i=0}^{k-1}\left(\frac{2n_c+i}{n_c+i}\right) \times \frac{1}{\left(1+{P_T}^{\frac{1}{n_c}}\right)^k} > 1 \qquad \text{(B.7)}
$$

The inequality in Eq. (B.7) holds only when the following condition holds:

$$
\begin{aligned}
& \prod_{i=0}^{k-1}\left(\frac{2n_c+i}{n_c+i}\right) \times \frac{1}{\left(1+{P_T}^{\frac{1}{n_c}}\right)^k} > 1 \\
\Leftrightarrow \quad & \prod_{i=0}^{k-1}\left(\frac{2n_c+i}{n_c+i}\right) > \left(1+{P_T}^{\frac{1}{n_c}}\right)^k \\
\Leftrightarrow \quad & {P_T}^{\frac{1}{n_c}} < \sqrt[k]{\prod_{i=0}^{k-1}\left(\frac{2n_c+i}{n_c+i}\right)} - 1 \\
\Leftrightarrow \quad & P_T < \left[\sqrt[k]{\prod_{i=0}^{k-1}\left(\frac{2n_c+i}{n_c+i}\right)} - 1\right]^{n_c} \qquad \text{(B.8)}
\end{aligned}
$$

The right-hand side of the inequality in Eq. (B.8), is a decreasing function with respect to n_c. This statement comes from the fact that a^x decreases with x when $0 < a < 1$.

Next, we show that the following condition holds:

$$\sqrt[k]{\prod_{i=0}^{k-1}\left(\frac{2n_c+i}{n_c+i}\right)} - 1 < 1 \tag{B.9}$$

We start by providing an upper bound for the term under the k-th square root in Eq. (B.9), *i.e.*:

$$\begin{aligned}
\prod_{i=0}^{k-1}\frac{2n_c+i}{n_c+i} &= \prod_{i=0}^{k-1}\left(\frac{2n_c+i+i-i}{n_c+i}\right) \\
&= \prod_{i=0}^{k-1}\left(\frac{2n_c+2i-i}{n_c+i}\right) \\
&= \prod_{i=0}^{k-1}\left(2-\frac{i}{n_c+i}\right) < \prod_{i=0}^{k-1} 2 = 2^k \\
\Leftrightarrow \prod_{i=0}^{k-1}\frac{2n_c+i}{n_c+i} &< 2^k
\end{aligned} \tag{B.10}$$

Using Eq. (B.10), we get:

$$\sqrt[k]{\prod_{i=0}^{k-1}\frac{2n_c+i}{n_c+i}} < \sqrt[k]{2^k} = 2 \tag{B.11}$$

By subtracting one on both sides of the inequality given in Eq. (B.11), we get:

$$\sqrt[k]{\prod_{i=0}^{k-1}\frac{2n_c+i}{n_c+i}} - 1 < 2 - 1 = 1 \tag{B.12}$$

Eq. (B.12) shows that the condition in Eq. (B.9) holds, and by that it shows that the right-hand side of Eq. (B.8) is a decreasing function with respect to n_c. Since the right-hand side of Eq. (B.8) is a decreasing function with respect to n_c, for any $n_c > 1$ the function will be greater than the its limit when n_c approaches infinity, which we express with the following equation:

$$P_T < \lim_{n_c\to\infty}\left[\sqrt[k]{\prod_{i=0}^{k-1}\left(\frac{2n_c+i}{n_c+i}\right)} - 1\right]^{n_c} < \left[\sqrt[k]{\prod_{i=0}^{k-1}\left(\frac{2n_c+i}{n_c+i}\right)} - 1\right]^{n_c} \tag{B.13}$$

To evaluate the limit of the right-hand side expression of Eq. (B.8), we make use of the following property. Any function $f(x)$ can be re-written as:

$$f(x) = e^{\ln f(x)} \tag{B.14}$$

The proof for Eq. (B.14) is straightforward by computing the natural logarithm for both sides of the expression given in Eq. (B.14), *i.e.*:

$$\begin{aligned} & f(x) = e^{\ln f(x)} \\ \Leftrightarrow \quad & \ln f(x) = \ln e^{\ln f(x)} \\ \Leftrightarrow \quad & \ln f(x) = \ln f(x) \ln e = \ln f(x) \end{aligned} \tag{B.15}$$

Let us denote with $f(n_c)$ the right-hand side of Eq. (B.8), *i.e.*:

$$f(n_c) = \left[\sqrt[k]{\prod_{i=0}^{k-1} \left(\frac{2n_c + i}{n_c + i} \right)} - 1 \right]^{n_c} \tag{B.16}$$

To evaluate the limit of $f(n_c)$ when n_c approaches infinity, we use the property given in Eq. (B.14), and thus we get:

$$\lim_{n_c \to \infty} f(n_c) = \lim_{n_c \to \infty} e^{\ln f(n_c)} = e^{\lim\limits_{n_c \to \infty} \ln f(n_c)} \tag{B.17}$$

By denoting the expression $\ln f(n_c)$ with $v(n_c)$, we get:

$$\lim_{n_c \to \infty} f(n_c) = e^{\lim\limits_{n_c \to \infty} v(n_c)} \tag{B.18}$$

Thus, to evaluate the limit of $f(n_c)$ when n_c approaches infinity, it is sufficient to evaluate the limit of $v(n_c)$ when n_c approaches infinity.

Given Eq. (B.16), we evaluate $v(n_c)$ as:

$$\begin{aligned} v(n_c) &= \ln \left[\sqrt[k]{\prod_{i=0}^{k-1} \left(\frac{2n_c + i}{n_c + i} \right)} - 1 \right]^{n_c} \\ &= n_c \ln \left(\sqrt[k]{\prod_{i=0}^{k-1} \left(\frac{2n_c + i}{n_c + i} \right)} - 1 \right) \\ &= \frac{\ln \left(\sqrt[k]{\frac{(2n_c+k-1)(2n_c+k-2)\cdots 2n_c}{(n_c+k-1)(n_c+k-2)\cdots n_c}} - 1 \right)}{\frac{1}{n_c}} \end{aligned}$$

$$= \frac{\ln\left(\sqrt[k]{\frac{P_k(n_c)}{Q_k(n_c)}} - 1\right)}{\frac{1}{n_c}} \tag{B.19}$$

In Eq. (B.19), we have introduced the notations $P_k(n_c)$ and $Q_k(n_c)$ to represent polynomial functions of n_c of order k. $P_k(n_c)$ and $Q_k(n_c)$ are evaluated with the following equations:

$$P_k(n_c) = \prod_{i=0}^{k-1}(2n_c + i) = \sum_{i=1}^{k}\left(b_i \times {n_c}^i\right) \tag{B.20}$$

$$Q_k(n_c) = \prod_{i=0}^{k-1}(n_c + i) = \sum_{i=1}^{k}\left(a_i \times {n_c}^i\right) \tag{B.21}$$

The following relation exists between the coefficients of the the polynomial functions $P_k(n_c)$ and $Q_k(n_c)$:

$$b_i = 2^k \times a_i,\ i = 1, 2, \ldots k \tag{B.22}$$

The relation between the coefficients of the the polynomial functions $P_k(n_c)$ and $Q_k(n_c)$ comes from the fact the $P_k(n_c) = Q_k(2n_c)$ (see Eq. (B.20) and Eq. (B.21)). Observe that the coefficient a_k (the highest order term coefficient) of the polynomial function $Q_k(n_c)$ is equal to one, *i.e.* $a_k = 1$, and this implies that $b_k = 2^k$ (the highest order term coefficient of $P_k(n_c)$). We represent these facts with the following equations:

$$a_k = 1 \tag{B.23}$$

$$b_k = 2^k \tag{B.24}$$

From Eq. (B.19), we observe that $v(n_c)$ is expressed as a ratio of two functions $v_1(n_c)$ and $v_2(n_c)$, where $v_1(n_c)$ and $v_2(n_c)$ are evaluated as:

$$v_1(n_c) = \ln\left(\sqrt[k]{\frac{P_k(n_c)}{Q_k(n_c)}} - 1\right) \tag{B.25}$$

$$v_2(n_c) = \frac{1}{n_c} \tag{B.26}$$

Therefore the limit of $v(n_c)$ when n_c approaches infinity can be evaluated as:

$$\lim_{n_c \to \infty} v(n_c) = \frac{\lim\limits_{n_c \to \infty} v_1(n_c)}{\lim\limits_{n_c \to \infty} v_2(n_c)} \tag{B.27}$$

Next, we evaluate the limits of $v_1(n_c)$ and $v_2(n_c)$ when n_c approaches infinity.

The limit of $v_1(n_c)$ when n_c approaches infinity is evaluated as:

$$\begin{aligned}
\lim_{n_c\to\infty} v_1(n_c) &= \lim_{n_c\to\infty} \ln\left(\sqrt[k]{\frac{P_k(n_c)}{Q_k(n_c)}} - 1\right) \\
&= \ln\left(\sqrt[k]{\lim_{n_c\to\infty} \frac{P_k(n_c)}{Q_k(n_c)}} - 1\right) \\
&= \ln\left(\sqrt[k]{\lim_{n_c\to\infty} \frac{b_k \times {n_c}^k \times p_0(n_c)}{a_k \times {n_c}^k \times q_0(n_c)}} - 1\right) \\
&= \ln\left(\sqrt[k]{\lim_{n_c\to\infty} \frac{b_k \times p_0(n_c)}{a_k \times q_0(n_c)}} - 1\right) \\
&= \ln\left(\sqrt[k]{\frac{b_k}{a_k} \frac{\lim_{n_c\to\infty} p_0(n_c)}{\lim_{n_c\to\infty} q_0(n_c)}} - 1\right) \\
&= \ln\left(\sqrt[k]{\frac{b_k}{a_k}} - 1\right) \\
&= \ln\left(\sqrt[k]{\frac{2^k}{1}} - 1\right) \text{ using Eq. (B.23) and Eq. (B.24)} \\
&= \ln\left(\sqrt[k]{2^k} - 1\right) \\
&= \ln(2-1) \\
&= \ln(1) \\
&= 0
\end{aligned} \tag{B.28}$$

The limit of $v_2(n_c)$ when n_c approaches infinity is evaluated as:

$$\begin{aligned}
\lim_{n_c\to\infty} v_2(n_c) &= \lim_{n_c\to\infty} \frac{1}{n_c} \\
&= \frac{1}{\lim_{n_c\to\infty} n_c} \\
&= \frac{1}{\infty} \\
&= 0
\end{aligned} \tag{B.29}$$

Given Eq. (B.28) and Eq. (B.29), we evaluate the limit of $v(n_c)$ according to Eq. (B.27) as:

$$\lim_{n_c\to\infty} v(n_c) = \frac{\lim_{n_c\to\infty} v_1(n_c)}{\lim_{n_c\to\infty} v_2(n_c)} = \frac{0}{0} \tag{B.30}$$

Note that the limit as evaluated in Eq. (B.30) is not defined. However, this allows to use the l'Hôpital's rule to evaluate the limit of $v(n_c)$ which states:

$$\lim_{n_c\to\infty} v(n_c) = \lim_{n_c\to\infty} \frac{v_1(n_c)}{v_2(n_c)} = \lim_{n_c\to\infty} \frac{v_1'(n_c)}{v_2'(n_c)} \tag{B.31}$$

where $v_1'(n_c)$ and $v_2'(n_c)$ in Eq. (B.31) are the first derivatives of $v_1(n_c)$ and $v_2(n_c)$, respectively.

Given Eq. (B.31), we evaluate the limit of $v(n_c)$ as:

$$\begin{aligned}
&\lim_{n_c\to\infty} v(n_c) = \\
&= \lim_{n_c\to\infty} \frac{\frac{1}{\left(\frac{P_k(n_c)}{Q_k(n_c)}\right)^{\frac{1}{k}-1}} \times \frac{1}{k} \times \left(\frac{P_k(n_c)}{Q_k(n_c)}\right)^{\frac{1}{k}-1} \times \frac{(P_k(n_c))'Q_k(n_c) - P_k(n_c)(Q_k(n_c))'}{[Q_k(n_c)]^2}}{-\frac{1}{{n_c}^2}} \\
&= -\frac{1}{k} \lim_{n_c\to\infty} \frac{{n_c}^2 \left[(P_k(n_c))' \times Q_k(n_c) - P_k(n_c) \times (Q_k(n_c))'\right]}{\left[P_k(n_c) - (P_k(n_c))^{1-\frac{1}{k}} \times (Q_k(n_c))^{\frac{1}{k}}\right] Q_k(n_c)} \\
&= -\frac{1}{k} \lim_{n_c\to\infty} \frac{{n_c}^2 \left[\tilde{P}_{k-1}(n_c) \times Q_k(n_c) - P_k(n_c) \times \tilde{Q}_{k-1}(n_c)\right]}{\left[P_k(n_c) - (P_k(n_c))^{1-\frac{1}{k}} \times (Q_k(n_c))^{\frac{1}{k}}\right] Q_k(n_c)}
\end{aligned} \tag{B.32}$$

In Eq. (B.32), we have introduced the notations $\tilde{P}_{k-1}(n_c)$ and $\tilde{Q}_{k-1}(n_c)$ to represent the first derivatives of $P_k(n_c)$ and $Q_k(n_c)$, respectively. The derivatives, $\tilde{P}_{k-1}(n_c)$ and $\tilde{Q}_{k-1}(n_c)$, of the polynomial functions of n_c of order k, *i.e.* $P_k(n_c)$ and $Q_k(n_c)$, are again polynomial functions of n_c, but the order of these functions is $k-1$. The polynomial functions $\tilde{P}_{k-1}(n_c)$ and $\tilde{Q}_{k-1}(n_c)$ are evaluated as:

$$\begin{aligned}
\tilde{P}_{k-1}(n_c) &= (P_k(n_c))' \\
&= \left(\sum_{i=1}^{k} b_i \times {n_c}^i\right)' \\
&= \sum_{i=1}^{k} \left(b_i \times {n_c}^i\right)' \\
&= \sum_{i=1}^{k} i \times b_i \times {n_c}^{i-1} \\
&= \sum_{i=0}^{k-1} \underbrace{(i+1) \times b_{i+1}}_{\tilde{b}_i} \times {n_c}^i
\end{aligned}$$

$$= \sum_{i=0}^{k-1} \tilde{b}_i \times {n_c}^i \tag{B.33}$$

$$\tilde{b}_i = (i+1) \times b_{i+1},\ i = 0, 1, \ldots k-1 \tag{B.34}$$

$$\begin{aligned}
\tilde{Q}_{k-1}(n_c) &= (Q_k(n_c))' \\
&= \left(\sum_{i=1}^{k} a_i \times {n_c}^i \right)' \\
&= \sum_{i=1}^{k} \left(a_i \times {n_c}^i \right)' \\
&= \sum_{i=1}^{k} i \times a_i \times {n_c}^{i-1} \\
&= \sum_{i=0}^{k-1} \underbrace{(i+1) \times a_{i+1}}_{\tilde{a}_i} \times {n_c}^i \\
&= \sum_{i=0}^{k-1} \tilde{a}_i \times {n_c}^i
\end{aligned} \tag{B.35}$$

$$\tilde{a}_i = (i+1) \times a_{i+1},\ i = 0, 1, \ldots k-1 \tag{B.36}$$

Next, we evaluate the expression of the numerator in Eq. (B.32). We proceed by introducing the notations $X_{2k-1}(n_c)$ and $Y_{2k-1}(n_c)$, which are defined as:

$$X_{2k-1}(n_c) = \tilde{P}_{k-1}(n_c) \times Q_k(n_c) \tag{B.37}$$

$$Y_{2k-1}(n_c) = P_k(n_c) \times \tilde{Q}_{k-1}(n_c) \tag{B.38}$$

We re-write Eq. (B.37) as:

$$\begin{aligned}
&X_{2k-1}(n_c) = \\
&= \tilde{P}_{k-1}(n_c) \times Q_k(n_c) \\
&= \left(\tilde{b}_{k-1} {n_c}^{k-1} + \tilde{b}_{k-2} {n_c}^{k-2} + \tilde{P}_{k-3}(n_c) \right) \times \left(a_k {n_c}^k + a_{k-1} {n_c}^{k-1} + Q_{k-2}(n_c) \right) \\
&= \tilde{b}_{k-1} a_k {n_c}^{2k-1} + \left[\tilde{b}_{k-1} a_{k-1} + \tilde{b}_{k-2} a_k \right] {n_c}^{2k-2} + X_{2k-3}(n_c)
\end{aligned} \tag{B.39}$$

Similarly, we re-write Eq. (B.38) as:

$$\begin{aligned}
&Y_{2k-1}(n_c) = \\
&= P_k(n_c) \times \tilde{Q}_{k-1}(n_c) \\
&= \left(b_k {n_c}^k + b_{k-1} {n_c}^{k-1} + P_{k-2}(n_c) \right) \times \left(\tilde{a}_{k-1} {n_c}^{k-1} + \tilde{a}_{k-2} {n_c}^{k-2} + \tilde{Q}_{k-3}(n_c) \right) \\
&= \tilde{a}_{k-1} b_k {n_c}^{2k-1} + \left[\tilde{a}_{k-1} b_{k-1} + \tilde{a}_{k-2} b_k \right] {n_c}^{2k-2} + Y_{2k-3}(n_c)
\end{aligned} \tag{B.40}$$

Using Eq. (B.39) and Eq. (B.40), we evaluate the following expression:

$$\begin{aligned} X_{2k-1}(n_c) - Y_{2k-1}(n_c) = & \left(\tilde{b}_{k-1}a_k - \tilde{a}_{k-1}b_k\right) {n_c}^{2k-1} \\ & + \left(\tilde{b}_{k-1}a_{k-1} + \tilde{b}_{k-2}a_k - \tilde{a}_{k-1}b_{k-1} + \tilde{a}_{k-2}b_k\right) {n_c}^{2k-2} \\ & + X_{2k-3}(n_c) - Y_{2k-3}(n_c) \\ = & \left(\tilde{b}_{k-1}a_k - \tilde{a}_{k-1}b_k\right) {n_c}^{2k-1} \\ & + \left(\tilde{b}_{k-1}a_{k-1} + \tilde{b}_{k-2}a_k - \tilde{a}_{k-1}b_{k-1} - \tilde{a}_{k-2}b_k\right) {n_c}^{2k-2} \\ & + Z_{2k-3}(n_c) \end{aligned} \tag{B.41}$$

Next, we evaluate the coefficients in front of the terms ${n_c}^{2k-1}$ and ${n_c}^{2k-2}$ in Eq. (B.41). To evaluate these coefficients, we use Eq. (B.22), Eq. (B.23), Eq. (B.24), Eq. (B.34) and Eq. (B.36).

The coefficient in front of the term ${n_c}^{2k-1}$ in Eq. (B.41) is evaluated as:

$$\tilde{b}_{k-1}a_k - \tilde{a}_{k-1}b_k = kb_ka_k - ka_kb_k = 0 \tag{B.42}$$

The coefficient in front of the term ${n_c}^{2k-2}$ in Eq. (B.41) is evaluated as:

$$\begin{aligned} & \tilde{b}_{k-1}a_{k-1} + \tilde{b}_{k-2}a_k - \tilde{a}_{k-1}b_{k-1} - \tilde{a}_{k-2}b_k = \\ = \; & kb_ka_{k-1} + (k-1)\, b_{k-1}a_k - ka_kb_{k-1} - (k-1)\, a_{k-1}b_k \\ = \; & k2^k a_{k-1} + (k-1)\, 2^{k-1}a_{k-1} - k2^{k-1}a_{k-1} - 2^k\,(k-1)\, a_{k-1} \\ = \; & 2^{k-1}a_{k-1}\,(2-1) \\ = \; & 2^{k-1}a_{k-1} \end{aligned} \tag{B.43}$$

Replacing the coefficients in front of the terms ${n_c}^{2k-1}$ and ${n_c}^{2k-2}$ in Eq. (B.41) with the expressions given in Eq. (B.42) and Eq. (B.43), we get:

$$\begin{aligned} X_{2k-1}(n_c) - Y_{2k-1}(n_c) &= 2^{k-1}a_{k-1}{n_c}^{2k-2} + Z_{2k-3}(n_c) \\ &= {n_c}^{2k-2}\left[2^{k-1}a_{k-1} + z_{-1}(n_c)\right] \end{aligned} \tag{B.44}$$

Observe in Eq. (B.44), the term $z_{-1}(n_c)$ is a polynomial function of n_c where the highest order is -1. Using Eq. (B.44), we re-write Eq. (B.32) as:

$$\lim_{n_c\to\infty} v(n_c) = -\frac{1}{k}\lim_{n_c\to\infty} \frac{{n_c}^{2k}\left[2^{k-1}a_{k-1} + z_{-1}(n_c)\right]}{\left[P_k(n_c) - \left(P_k(n_c)\right)^{1-\frac{1}{k}} \times \left(Q_k(n_c)\right)^{\frac{1}{k}}\right] Q_k(n_c)} \tag{B.45}$$

Finally, we evaluate the limit of Eq. (B.45) when n_c approaches infinity, by

using Eq. (B.22), Eq. (B.24) and Eq. (B.23), as:

$$
\begin{aligned}
&\lim_{n_c\to\infty} v(n_c) = \\
&= -\frac{1}{k}\lim_{n_c\to\infty}\frac{{n_c}^{2k}\left[2^{k-1}a_{k-1}+z_{-1}(n_c)\right]}{\left[b_k{n_c}^k p_0(n_c)-\left(b_k{n_c}^k p_0(n_c)\right)^{1-\frac{1}{k}}\left(a_k{n_c}^k q_0(n_c)\right)^{\frac{1}{k}}\right]a_k{n_c}^k q_0(n_c)} \\
&= -\frac{1}{k}\lim_{n_c\to\infty}\frac{{n_c}^{2k}\left[2^{k-1}a_{k-1}+z_{-1}(n_c)\right]}{\left[2^k{n_c}^k p_0(n_c)-\left(2^k{n_c}^k p_0(n_c)\right)^{1-\frac{1}{k}}\left({n_c}^k q_0(n_c)\right)^{\frac{1}{k}}\right]{n_c}^k q_0(n_c)} \\
&= -\frac{1}{k}\lim_{n_c\to\infty}\frac{{n_c}^{2k}\left[2^{k-1}a_{k-1}+z_{-1}(n_c)\right]}{\left[2^k{n_c}^k p_0(n_c)-2^{k-1}{n_c}^{k-1}\left(p_0(n_c)\right)^{1-\frac{1}{k}}n_c\left(q_0(n_c)\right)^{\frac{1}{k}}\right]{n_c}^k q_0(n_c)} \\
&= -\frac{1}{k}\lim_{n_c\to\infty}\frac{{n_c}^{2k}\left[2^{k-1}a_{k-1}+z_{-1}(n_c)\right]}{\left[2^k{n_c}^k p_0(n_c)-2^{k-1}{n_c}^k\left(p_0(n_c)\right)^{1-\frac{1}{k}}\left(q_0(n_c)\right)^{\frac{1}{k}}\right]{n_c}^k q_0(n_c)} \\
&= -\frac{1}{k}\lim_{n_c\to\infty}\frac{{n_c}^{2k}\left[2^{k-1}a_{k-1}+z_{-1}(n_c)\right]}{{n_c}^{2k}\left[2^k p_0(n_c)-2^{k-1}\left(p_0(n_c)\right)^{1-\frac{1}{k}}\left(q_0(n_c)\right)^{\frac{1}{k}}\right]q_0(n_c)} \\
&= -\frac{1}{k}\lim_{n_c\to\infty}\frac{2^{k-1}a_{k-1}+z_{-1}(n_c)}{\left[2^k p_0(n_c)-2^{k-1}\left(p_0(n_c)\right)^{1-\frac{1}{k}}\left(q_0(n_c)\right)^{\frac{1}{k}}\right]q_0(n_c)} \\
&= -\frac{1}{k}\frac{2^{k-1}a_{k-1}+\lim_{n_c\to\infty} z_{-1}(n_c)}{\left[2^k\lim_{n_c\to\infty}p_0(n_c)-2^{k-1}\left(\lim_{n_c\to\infty}p_0(n_c)\right)^{1-\frac{1}{k}}\left(\lim_{n_c\to\infty}q_0(n_c)\right)^{\frac{1}{k}}\right]\lim_{n_c\to\infty}q_0(n_c)} \\
&= -\frac{1}{k}\frac{2^{k-1}a_{k-1}}{\left[2^k-2^{k-1}\right]} \\
&= -\frac{a_{k-1}}{k}
\end{aligned}
\tag{B.46}
$$

From Eq. (B.46), one sees that the limit of $v(n_c)$ when n_c approaches infinity, depends on the coefficient a_{k-1} of the k-th order polynomial $Q_k(n_c)$. Next, we provide an expression on how to evaluate a_{k-1}.

As discussed earlier, a_{k-1} is the coefficient of the term ${n_c}^{k-1}$ in the polynomial $Q_k(n_c)$, presented in Eq. (B.21). Further more, let us use the notation $a_{k-1}^{(k)}$ to denote the coefficient a_{k-1} of a k-th order polynomial. From Eq. (B.21),

we can express $Q_k(n_c)$ as:

$$\begin{aligned} Q_k(n_c) &= \prod_{i=0}^{k-1}(n_c+i) \\ &= (n_c+k-1)\times\prod_{i=0}^{k-2}(n_c+i) \\ &= (n_c+k-1)\times Q_{k-1}(n_c) \end{aligned} \tag{B.47}$$

The polynomial $Q_{k-1}(n_c)$ can be expressed as:

$$\begin{aligned} Q_{k-1}(n_c) &= \sum_{i=1}^{k-1} a_i^{(k-1)} {n_c}^i \\ &= a_{k-1}^{(k-1)} {n_c}^{k-1} + a_{k-2}^{(k-1)} {n_c}^{k-2} + Q_{k-3}(n_c) \end{aligned} \tag{B.48}$$

Important to note, is that the coefficient $a_{k-1}^{(k-1)}$ is always 1, for any $k \geq 1$.

Replacing $Q_{k-1}(n_c)$ in Eq. (B.47) with the expression in Eq. (B.48) leads to:

$$\begin{aligned} Q_k(n_c) &= (n_c+k-1)\times Q_{k-1}(n_c) \\ &= (n_c+k-1)\times\left[a_{k-1}^{(k-1)} {n_c}^{k-1} + a_{k-2}^{(k-1)} {n_c}^{k-2} + Q_{k-3}(n_c)\right] \\ &= {n_c}^k + \left[k-1+a_{k-2}^{(k-1)}\right] {n_c}^{k-1} + Q_{k-2}(n_c) \end{aligned} \tag{B.49}$$

From Eq. (B.48), we obtain the following recursive formula that computes the coefficient $a_{k-1}^{(k)}$:

$$\begin{aligned} a_{k-1}^{(k)} &= k-1+a_{k-2}^{(k-1)},\ k>1 \\ a_{k-1}^{(k)} &= a_0^{(1)} = 0,\ k=1 \end{aligned} \tag{B.50}$$

The recursive formula given in Eq. (B.50) has the following solution:

$$a_{k-1}^{(k)} = \frac{k\,(k-1)}{2} \tag{B.51}$$

We prove that Eq. (B.51) is the solution for the recursive formula given in Eq. (B.50) by using mathematical induction.

As the first step of mathematical induction is to check whether the assumption is correct, let us assume $k=3$. For $k=3$, according to Eq. (B.51), $a_2^{(3)} = 3$. The polynomial $Q_3(n_c)$ is evaluated as:

$$Q_3(n_c) = \prod_{i=0}^{2}(n_c+i) = {n_c}^3 + 3{n_c}^2 + 2n_c \tag{B.52}$$

From Eq. (B.52), we note that the coefficient of the term n_c^2 is 3, and the same result is obtained when Eq. (B.51) is used. This shows that the assumption is correct for a base case.

Next, we apply the inductive step. Let us assume $k = k^\dagger + 1$, and assume that Eq. (B.51) is satisfied for $k = k^\dagger$, *i.e.*:

$$a_{k^\dagger - 1}^{(k^\dagger)} = \frac{k^\dagger \left(k^\dagger - 1\right)}{2} \tag{B.53}$$

The recursive formula, given in Eq. (B.50), for $k = k^\dagger + 1$ is evaluated as:

$$a_{k^\dagger}^{(k^\dagger + 1)} = k^\dagger + a_{k^\dagger - 1}^{(k^\dagger)} \tag{B.54}$$

Using Eq. (B.53) and replacing the appropriate term in Eq. (B.54) we get:

$$\begin{aligned} a_{k^\dagger}^{(k^\dagger + 1)} &= k^\dagger + a_{k^\dagger - 1}^{(k^\dagger)} \\ &= k^\dagger + \frac{k^\dagger \left(k^\dagger - 1\right)}{2} \\ &= \frac{k^\dagger \left(k^\dagger - 1 + 2\right)}{2} \\ &= \frac{k^\dagger \left(k^\dagger + 1\right)}{2} \end{aligned} \tag{B.55}$$

Observe that the expression in Eq. (B.55) is the same expression that we would obtain if we use Eq. (B.51) for $k = k^\dagger + 1$. By this, we prove that Eq. (B.51) is the solution of the recursive formula given in Eq. (B.50).

By replacing the expression of Eq. (B.51) in Eq. (B.46), we obtain the limit of $v(n_c)$ when n_c approaches infinity as:

$$\lim_{n_c \to \infty} v(n_c) = -\frac{a_{k-1}}{k} = -\frac{k\,(k-1)}{2k} = -\frac{k-1}{2} \tag{B.56}$$

Finally, by using Eq. (B.56), we evaluate the limit of $f(n_c)$, given in Eq. (B.18), when n_c approaches infinity as:

$$\begin{aligned} \lim_{n_c \to \infty} f(n_c) &= e^{\lim\limits_{n_c \to \infty} v(n_c)} \\ &= e^{-\frac{k-1}{2}} \end{aligned} \tag{B.57}$$

Thus for any k, which satisfies the following condition (see Eq. (B.13)):

$$P_T < e^{-\frac{k-1}{2}} \tag{B.58}$$

implies that for any $n_{c_1} > n_{c_2}$, $p_{n_{c_1}}(k) > p_{n_{c_2}}(k)$.

As seen in Eq. (B.57), the limit of $f(n_c)$ when n_c approaches infinity is a decreasing function with respect to k. That means, if the condition in Eq. (B.58) is satisfied for a given $k^{\dagger}$, the same condition will be satisfied for any $k \leq k^{\dagger}$. Furthermore, for any given $0 < P_T < 1$, we can find K_l (the largest value of k that satisfies the condition in Eq. (B.58)), such that the following condition will hold:

$$p_{n_{c_1}}(k) > p_{n_{c_2}}(k), \ \forall k \in [1, K_l] \tag{B.59}$$

for any n_{c_1} and n_{c_2}, such that $n_{c_1} > n_{c_2}$.

Another important observation is that **Lemma 1** is a special case of **Lemma 2**. Using **Lemma 2** for the specific case when $k = 1$, leads to the same results as given by **Lemma 1**. The limit of $f(n_c)$ when n_c approaches infinity, for $k = 1$, is evaluated, according to Eq. (B.57), as one. Due to the fact that $0 < P_T < 1$, the condition in Eq. (B.58) will always be satisfied for $k = 1$, which means that for any $n_{c_1} > n_{c_2}$, $p_{n_{c_1}}(1) > p_{n_{c_2}}(1)$, which was stated in **Lemma 1**. However, we have provided the proofs for **Lemma 1** and **2** in two completely different ways, and yet shown that the same results are obtained.

Next, we provide the proof for **Lemma 3**, which states that $p_{n_{c_1}}(k) < p_{n_{c_2}}(k)$ for any $n_{c_1} > n_{c_2}$ and any $k \geq K_u$, such that $P_T > \sqrt[k]{k+1} - 1$. Again, the proof is based on the ratio:

$$\frac{p_{2n_c}(k)}{p_{n_c}(k)} = \prod_{i=0}^{k-1} \left(\frac{2n_c + i}{n_c + i} \right) \times \frac{1}{\left(1 + P_T{}^{\frac{1}{n_c}}\right)^k} \tag{B.60}$$

When the ratio in Eq. (B.60) is less than one, implies that $p_{2n_c}(k) < p_{n_c}(k)$, *i.e.* for $n_{c_1} = 2n_c > n_{c_2} = n_c$ we get $p_{n_{c_1}}(k) < p_{n_{c_2}}(k)$. Such inequality would hold only when the following condition is satisfied:

$$P_T > \left[\sqrt[k]{\prod_{i=0}^{k-1} \left(\frac{2n_c + i}{n_c + i} \right)} - 1 \right]^{n_c} \tag{B.61}$$

While proving **Lemma 2**, we have shown that the right-hand side expression of Eq. (B.61) is a decreasing function with respect to n_c. As $n_c \geq 1$, the largest value of the right-hand side expression of Eq. (B.61) is reached at $n_c = 1$. This allows us to write the following inequality:

$$P_T > \sqrt[k]{\prod_{i=0}^{k-1} \left(\frac{2 + i}{1 + i} \right)} - 1 > \left[\sqrt[k]{\prod_{i=0}^{k-1} \left(\frac{2n_c + i}{n_c + i} \right)} - 1 \right]^{n_c} \tag{B.62}$$

Furthermore, the largest value of the right-hand side expression of Eq. (B.61) can be expressed as:

$$\begin{aligned}
&\sqrt[k]{\prod_{i=0}^{k-1}\left(\frac{2+i}{1+i}\right)}-1 \\
&=\sqrt[k]{\tfrac{(2+k-1)(2+k-2)\cdots 2}{(1+k-1)(1+k-2)\cdots 1}}-1 \\
&=\sqrt[k]{\tfrac{(k+1)(k)\cdots 2}{(k)(k-1)\cdots 1}\tfrac{1}{1}}-1 \\
&=\sqrt[k]{\tfrac{(k+1)(k)\cdots 2\cdot 1}{(k)(k-1)\cdots 1}}-1 \\
&=\sqrt[k]{\tfrac{(k+1)!}{(k)!}}-1 \\
&=\sqrt[k]{k+1}-1
\end{aligned} \tag{B.63}$$

Eq. (B.63) allows us to re-write the condition in Eq. (B.62) as:

$$P_T > \sqrt[k]{k+1}-1 > \left[\sqrt[k]{\prod_{i=0}^{k-1}\left(\frac{2n_c+i}{n_c+i}\right)}-1\right]^{n_c} \tag{B.64}$$

Hence, for any k that satisfies the following condition:

$$P_T > \sqrt[k]{k+1}-1 \tag{B.65}$$

for any $n_{c_1} > n_{c_2}$, $p_{n_{c_1}}(k) < p_{n_{c_2}}(k)$.

Important to note is that the condition in Eq. (B.65) can be satisfied if and only if $k > 1$, given that $0 < P_T < 1$. Evaluating the condition in Eq. (B.65) for $k = 1$, states that $P_T > 1$. However, as P_T is a probability, it can never be larger than 1, which shows that at $k = 1$, for any $n_{c_1} > n_{c_2}$, $p_{n_{c_1}}(k)$ cannot be less than $p_{n_{c_2}}(k)$. We showed while proving **Lemma 1** and **2** that for any $n_{c_1} > n_{c_2}$, $p_{n_{c_1}}(1) > p_{n_{c_2}}(1)$.

Another important observation is that for any $k > 1 \Leftrightarrow k \geq 2$ the right-hand side of Eq. (B.65) is a deceasing function with respect to k. To show this, let us denote the right-hand side of Eq. (B.65) with $f(k)$, *i.e.*:

$$f(k) = \sqrt[k]{k+1}-1 \tag{B.66}$$

We show that $f(k)$ is decreasing with respect to k, for any $k \geq 2$, by computing its first derivative and comparing it with zero. The first derivative of

$f(k)$ is evaluated as:

$$\begin{aligned} f'(k) &= \left(\sqrt[k]{k+1}-1\right)' \\ &= \left(\sqrt[k]{k+1}\right)' \\ &= \left(e^{\ln(k+1)^{\frac{1}{k}}}\right)' \\ &= \left(e^{\frac{1}{k}\ln(k+1)}\right)' \\ &= e^{\frac{1}{k}\ln(k+1)}\left[-\frac{1}{k^2}\ln(k+1)+\frac{1}{k(k+1)}\right] \\ &= e^{\frac{1}{k}\ln(k+1)}\left[\frac{(1-\ln(k+1))k-\ln(k+1)}{k(k+1)}\right] \end{aligned} \tag{B.67}$$

The first derivative of $f(k)$, as given in Eq. (B.67), can be negative (less than zero) only if the term in the numerator is negative, *i.e.*:

$$f'(k) < 0 \Leftrightarrow (1-\ln(k+1))k - \ln(k+1) < 0 \tag{B.68}$$

The term $-\ln(k+1)$ is negative (less than zero) for any $k > 0$, and further more $1-\ln(k+1)$ is negative for any $k > e-1 \Leftrightarrow k \geq 2$. From this we conclude that $f'(k) < 0$, $k \geq 2$, which implies that $f(k)$ decreases with $k \geq 2$.

Given that the right-hand side of Eq. (B.65) is decreasing with k, implies that if for a given $k^\dagger$ the condition in Eq. (B.65) is satisfied, the same condition will be satisfied for any $k \geq k^\dagger$. Furthermore, for any given $0 < P_T < 1$, we can find K_u (the lowest k which satisfies the condition in Eq. (B.65)), such that the following condition will hold:

$$p_{n_{c_1}}(k) < p_{n_{c_2}}(k),\ \forall\ k\ \in [K_u, \infty) \tag{B.69}$$

for any n_{c_1} and n_{c_2}, such that $n_{c_1} > n_{c_2}$.

Next, we construct the proof for **Theorem 3**, which states that for any given $n_{c_1} > n_{c_2}$, and for a given fixed number of re-executions $k^\dagger$, the following relation holds $\lambda_{n_{c_1}}(k^\dagger) > \lambda_{n_{c_2}}(k^\dagger)$. To construct the proof, we use **Lemma 1**, **2** and **3**, from which we obtain that for any $0 < P_T < 1$ there exist K_l and K_u, such that for any given $n_{c_1} > n_{c_2}$ the following inequalities hold:

$$p_{n_{c_1}}(k) > p_{n_{c_2}}(k),\ 1 \leq k \leq K_l \tag{B.70}$$

$$p_{n_{c_1}}(k) < p_{n_{c_2}}(k),\ K_u \leq k < \infty \tag{B.71}$$

We show that for any given $k^\dagger$, irrespective of P_T (hence, irrespective of K_l and K_u), $\lambda_{n_{c_1}}(k^\dagger) > \lambda_{n_{c_2}}(k^\dagger)$ as long as $n_{c_1} > n_{c_2}$.

We find the relation between $\lambda_{n_{c_1}}(k^\dagger)$ and $\lambda_{n_{c_2}}(k^\dagger)$, for given $n_{c_1} > n_{c_2}$ while assuming the following three cases:

- *Case* I: $k^\dagger \leq K_l$;
- *Case* II: $k^\dagger \geq K_u$; and
- *Case* III: $K_l < k^\dagger < K_u$;

Case I. Given n_{c_1} and n_{c_2} such that $n_{c_1} > n_{c_2}$, and $k^\dagger \leq K_l$, we show that $\lambda_{n_{c_1}}(k^\dagger) > \lambda_{n_{c_2}}(k^\dagger)$. The proof is provided with the following set of equations:

$$\begin{aligned}
\lambda_{n_{c_1}}(k^\dagger) &= p_{n_{c_1}}(0) + \sum_{k=1}^{k^\dagger} p_{n_{c_1}}(k) \\
&= {P_T}^2 + \sum_{k=1}^{k^\dagger} p_{n_{c_1}}(k) \\
&> \underbrace{{P_T}^2 + \sum_{k=1}^{k^\dagger} p_{n_{c_2}}(k)}_{\lambda_{n_{c_2}}(k^\dagger)} \text{ using Eq. (B.70)} \\
&> \lambda_{n_{c_2}}(k^\dagger)
\end{aligned} \tag{B.72}$$

Case II. Given n_{c_1} and n_{c_2} such that $n_{c_1} > n_{c_2}$, and $k^\dagger \geq K_u$, we show that $\lambda_{n_{c_1}}(k^\dagger) > \lambda_{n_{c_2}}(k^\dagger)$. To proof the given inequality we use the following equation:

$$\begin{aligned}
\sum_{k=0}^{\infty} p_{n_c}(k) &= 1 \\
\sum_{k=0}^{k^\dagger} p_{n_c}(k) + \sum_{k=k^\dagger+1}^{\infty} p_{n_c}(k) &= 1 \\
\sum_{k=0}^{k^\dagger} p_{n_c}(k) &= 1 - \sum_{k=k^\dagger+1}^{\infty} p_{n_c}(k) \\
\lambda_{n_c}(k^\dagger) &= 1 - \sum_{k=k^\dagger+1}^{\infty} p_{n_c}(k)
\end{aligned} \tag{B.73}$$

Using Eq. (B.73), we express $\lambda_{n_{c_1}}(k^\dagger)$ and $\lambda_{n_{c_2}}k^\dagger)$ as:

$$\lambda_{n_{c_1}}(k^\dagger) = 1 - \sum_{k=k^\dagger+1}^{\infty} p_{n_{c_1}}(k) \tag{B.74}$$

$$\lambda_{n_{c_2}}(k^\dagger) \quad = \quad 1 - \sum_{k=k^\dagger+1}^{\infty} p_{n_{c_2}}(k) \tag{B.75}$$

Using Eq. (B.71) and the fact that $k^\dagger \geq K_u$, the following equation holds:

$$\sum_{k=k^\dagger+1}^{\infty} p_{n_{c_1}}(k) < \sum_{k=k^\dagger+1}^{\infty} p_{n_{c_2}}(k) \tag{B.76}$$

By further re-working the expression given in Eq. (B.76), we get:

$$\begin{aligned}
& \sum_{k=k^\dagger+1}^{\infty} p_{n_{c_1}}(k) < \sum_{k=k^\dagger+1}^{\infty} p_{n_{c_2}}(k) \\
\Leftrightarrow \quad & - \sum_{k=k^\dagger+1}^{\infty} p_{n_{c_1}}(k) > - \sum_{k=k^\dagger+1}^{\infty} p_{n_{c_2}}(k) \\
\Leftrightarrow \quad & 1 - \sum_{k=k^\dagger+1}^{\infty} p_{n_{c_1}}(k) > 1 - \sum_{k=k^\dagger+1}^{\infty} p_{n_{c_2}}(k) \\
\Leftrightarrow \quad & \lambda_{n_{c_1}}(k^\dagger) > \lambda_{n_{c_2}}(k^\dagger)
\end{aligned} \tag{B.77}$$

As seen from Eq. (B.77), the inequality $\lambda_{n_{c_1}}(k^\dagger) > \lambda_{n_{c_2}}(k^\dagger)$ holds when $k^\dagger \geq K_u$.

Case III. Given n_{c_1} and n_{c_2} such that $n_{c_1} > n_{c_2}$, and $K_l < k^\dagger < K_u$, we show that $\lambda_{n_{c_1}}(k^\dagger) > \lambda_{n_{c_2}}(k^\dagger)$. Before we proceed, let us elaborate on the relation between $p_{n_{c_1}}(k)$ and $p_{n_{c_2}}(k)$ for $K_l < k < K_u$. We have shown that $p_{n_{c_1}}(k) > p_{n_{c_2}}(k)$ for any $n_{c_1} > n_{c_2}$ when $k < K_l$, and we have shown that $p_{n_{c_1}}(k) < p_{n_{c_2}}(k)$ for any $n_{c_1} > n_{c_2}$ when $k > K_u$. For a given $K_l < k < K_u$, there exists an $n_c^\dagger$ such that for any $n_{c_2} < n_{c_1} < n_c^\dagger$ the following relation holds $p_{n_{c_1}}(k) > p_{n_{c_2}}(k)$, and for any $n_{c_1} > n_{c_2} > n_c^\dagger$ the following relation holds $p_{n_{c_1}}(k) < p_{n_{c_2}}(k)$. However, there is no evident relation between $p_{n_{c_1}}(k)$ and $p_{n_{c_2}}(k)$ when $n_{c_1} > n_c^\dagger$ and $n_{c_2} < n_c^\dagger$. Nonetheless, for given n_{c_1} and n_{c_2}, such that $n_{c_1} > n_{c_2}$, the fact that $p_{n_{c_1}}(k)$ is certainly greater than $p_{n_{c_2}}(k)$ for any $k \leq K_l$, and $p_{n_{c_1}}(k)$ is certainly less than $p_{n_{c_2}}(k)$ for any $k \geq K_u$, implies that there exists a $k^\ddagger$, where $K_l < k^\ddagger < K_u$, such that for the given n_{c_1} and n_{c_2}, the following relations hold:

$$p_{n_{c_1}}(k) > p_{n_{c_2}}(k), \; K_l \leq k \leq k^\ddagger \tag{B.78}$$

$$p_{n_{c_1}}(k) < p_{n_{c_2}}(k), \; k^\ddagger < k \leq K_u \tag{B.79}$$

Therefore, we split the case when $K_l < k^\dagger < K_u$ into two sub-cases:

- *Case* III.A.: $K_l < k^\dagger \leq k^\ddagger$;
- *Case* III.B.: $k^\ddagger < k^\dagger \leq K_u$;

Case III.A. This case is very similar to *Case* I. The following equations show that $\lambda_{n_{c_1}}(k^\dagger) > \lambda_{n_{c_2}}(k^\dagger)$ when $K_l < k^\dagger < k^\ddagger$:

$$\begin{aligned}
\lambda_{n_{c_1}}(k^\dagger) =& p_{n_{c_1}}(0) + \sum_{k=1}^{k^\dagger} p_{n_{c_1}}(k) \\
=& {P_T}^2 + \sum_{k=1}^{k^\dagger} p_{n_{c_1}}(k) \\
=& {P_T}^2 + \sum_{k=1}^{K_l} p_{n_{c_1}}(k) + \sum_{k=K_l+1}^{k^\dagger} p_{n_{c_1}}(k) \\
>& \underbrace{{P_T}^2 + \sum_{k=1}^{K_l} p_{n_{c_2}}(k) + \sum_{k=K_l+1}^{k^\dagger} p_{n_{c_2}}(k)}_{\lambda_{n_{c_2}}(k^\dagger)} \text{ using Eq. (B.70) and Eq. (B.78)} \\
>& \lambda_{n_{c_2}}(k^\dagger) \qquad \text{(B.80)}
\end{aligned}$$

Case III.B. This case is very similar to *Case* II. First, using Eq. (B.79) and $k^\ddagger < k\dagger \leq k \leq K_u$, we get:

$$\sum_{k=k^\dagger+1}^{K_u} p_{n_{c_1}}(k) < \sum_{k=k^\dagger+1}^{K_u} p_{n_{c_2}}(k) \qquad \text{(B.81)}$$

Second, using Eq. (B.71) and $K_u \leq k < \infty$, we get:

$$\sum_{k=K_u+1}^{\infty} p_{n_{c_1}}(k) < \sum_{k=K_u+1}^{\infty} p_{n_{c_2}}(k) \qquad \text{(B.82)}$$

By adding Eq. (B.81) and Eq. (B.82), we get:

$$\begin{aligned}
& \sum_{k=k^\dagger+1}^{K_u} p_{n_{c_1}}(k) + \sum_{k=K_u+1}^{\infty} p_{n_{c_1}}(k) < \sum_{k=k^\dagger+1}^{K_u} p_{n_{c_2}}(k) + \sum_{k=K_u+1}^{\infty} p_{n_{c_2}}(k) \\
\Leftrightarrow \quad & \sum_{k=k^\dagger+1}^{\infty} p_{n_{c_1}}(k) < \sum_{k=k^\dagger+1}^{\infty} p_{n_{c_2}}(k) \\
\Leftrightarrow \quad & -\sum_{k=k^\dagger+1}^{\infty} p_{n_{c_1}}(k) > -\sum_{k=k^\dagger+1}^{\infty} p_{n_{c_2}}(k)
\end{aligned}$$

$$\Leftrightarrow \quad 1 - \sum_{k=k^{\dagger}+1}^{\infty} p_{n_{c_1}}(k) > 1 - \sum_{k=k^{\dagger}+1}^{\infty} p_{n_{c_2}}(k)$$

$$\Leftrightarrow \quad \lambda_{n_{c_1}}(k^{\dagger}) > \lambda_{n_{c_2}}(k^{\dagger})$$

To conclude, for any $k^{\dagger}$, irrespective of K_l and K_u (hence irrespective of P_T), we have shown that $\lambda_{n_{c_1}}(k^{\dagger}) > \lambda_{n_{c_2}}(k^{\dagger})$ as long as $n_{c_1} > n_{c_2}$ (stated in **Theorem 3**). ■

Appendix C

In this section we provide the proof for **Theorem 4**.

Theorem 4. *For any k there exists a limit $\bar{\lambda}_k$, such that for any n_c the following condition holds:*

$$\lambda_{n_c}(k) \leq \bar{\lambda}_k$$

and $\bar{\lambda}_k$ is defined as:

$$\bar{\lambda}_k = {P_T}^2 \sum_{i=0}^{k} \frac{(-2\ln(P_T))^i}{i!}$$

Proof. This theorem is a consequence of **Theorem 3**, and the fundamental property of the LoC, *i.e.*

$$\lim_{k\to\infty} \lambda_{n_c}(k) = \sum_{k=0}^{\infty} p_{n_c}(k) = 1 \tag{C.1}$$

According to Eq. (C.1), the LoC is equal to one when k approaches infinity for any given n_c, and the proof was presented in Appendix A. Since **Theorem 3** states that for a fixed k the LoC, $\lambda_{n_c}(k)$, increases along with n_c, to satisfy Eq. (C.1) it is necessary that for a fixed k there exists a limit of $\lambda_{n_c}(k)$ when n_c approaches infinity. We denote this limit with $\bar{\lambda}_{n_c}(k)$. Next, we derive the expression to compute $\bar{\lambda}_{n_c}(k)$.

The limit $\bar{\lambda}_{n_c}(k)$ is defined as:

$$\begin{aligned}
\bar{\lambda}_{n_c}(k) &= \lim_{n_c\to\infty} \lambda_{n_c}(k) \\
&= \lim_{n_c\to\infty} \sum_{i=0}^{k} p_{n_c}(i) \\
&= \sum_{i=0}^{k} \lim_{n_c\to\infty} p_{n_c}(i)
\end{aligned} \tag{C.2}$$

As shown in the right-hand side of Eq. (C.2), to obtain $\bar{\lambda}_{n_c}(k)$ it is required to compute the limit of $p_{n_c}(i)$, $i \in [0,k]$, when n_c approaches infinity.

The limit of $p_{n_c}(k)$ when n_c approaches infinity, is defined as:

$$\begin{aligned}
\lim_{n_c\to\infty} p_{n_c}(k) &= \lim_{n_c\to\infty} \binom{n_c+k-1}{k} \times P_\epsilon^{n_c} \times (1-P_\epsilon)^k \\
&= \lim_{n_c\to\infty} \binom{n_c+k-1}{k} \times P_T^2 \times (1-{P_T}^{\frac{2}{n_c}})^k \\
&= \lim_{n_c\to\infty} \frac{P_k(n_c)}{k!} \times P_T^2 \times (1-{P_T}^{\frac{2}{n_c}})^k \\
&= \lim_{n_c\to\infty} \frac{{P_T}^2}{k!} \times P_k(n_c) \times (1-{P_T}^{\frac{2}{n_c}})^k \qquad \text{(C.3)}
\end{aligned}$$

The term $P_k(n_c)$ in Eq. (C.3) represents a polynomial of order k with respect to n_c, *i.e.*

$$P_k(n_c) = \sum_{i=1}^{k} a_i {n_c}^i \qquad \text{(C.4)}$$

Furthermore, the coefficient a_k for the polynomial $P_k(n_c)$ is $a_k = 1$. Using Eq. (C.3) and Eq. (C.4) we get:

$$\begin{aligned}
\lim_{n_c\to\infty} p_{n_c}(k) &= \lim_{n_c\to\infty} \frac{{P_T}^2}{k!} \times \sum_{i=1}^{k} a_i {n_c}^i \times (1-{P_T}^{\frac{2}{n_c}})^k \\
&= \frac{{P_T}^2}{k!} \times \lim_{n_c\to\infty} \sum_{i=1}^{k} a_i {n_c}^i \times (1-{P_T}^{\frac{2}{n_c}})^k \\
&= \frac{{P_T}^2}{k!} \times \sum_{i=1}^{k} a_i \lim_{n_c\to\infty} {n_c}^i \times (1-{P_T}^{\frac{2}{n_c}})^k \\
&= \frac{{P_T}^2}{k!} \times \sum_{i=1}^{k} a_i \lim_{n_c\to\infty} \left(n_c \times (1-{P_T}^{\frac{2}{n_c}})^{\frac{k}{i}}\right)^i \\
&= \frac{{P_T}^2}{k!} \times \sum_{i=1}^{k} a_i \left(\lim_{n_c\to\infty} \underbrace{n_c \times (1-{P_T}^{\frac{2}{n_c}})^{\frac{k}{i}}}_{f(n_c)}\right)^i \\
&= \frac{{P_T}^2}{k!} \times \sum_{i=1}^{k} a_i \left(\lim_{n_c\to\infty} f(n_c)\right)^i \qquad \text{(C.5)}
\end{aligned}$$

In Eq. (C.5) we have introduced the term $f(n_c)$ which is defined as:

$$f(n_c) = \begin{cases} n_c \times (1-{P_T}^{\frac{2}{n_c}}) & , \text{ if } i = k \\ n_c \times (1-{P_T}^{\frac{2}{n_c}})^b & , \text{ if } i < k \end{cases} \qquad \text{(C.6)}$$

The parameter b used in Eq. (C.6), is defined as $b = \frac{k}{i}$, and it is always larger than one (observe that the parameter b in Eq. (C.6) is used only for the case $i < k$). As the expression $f(n_c)$ is defined for two different cases, *i.e.* when $i = k$ and when $i < k$, we evaluate the limit of $f(n_c)$ when n_c approaches infinity for both cases.

For the first case, *i.e.* when $i = k$, the limit of $f(n_c)$ when n_c approaches infinity is evaluated as:

$$\lim_{n_c \to \infty} f(n_c) = \lim_{n_c \to \infty} n_c \times (1 - {P_T}^{\frac{2}{n_c}}) \tag{C.7}$$

By directly replacing n_c with infinity in Eq. (C.7) we get that the limit is undefined as we need to multiply infinity with zero. However, this allows us to use the l'Hôpital's rule to evaluate the limit of $f(n_c)$. By using l'Hôpital's rule we get:

$$\begin{aligned}
\lim_{n_c \to \infty} f(n_c) &= \lim_{n_c \to \infty} \frac{1 - {P_T}^{\frac{2}{n_c}}}{\frac{1}{n_c}} \\
&= \lim_{n_c \to \infty} \frac{-\ln(P_T) \times {P_T}^{\frac{2}{n_c}} \times \frac{-2}{{n_c}^2}}{\frac{-1}{{n_c}^2}} \\
&= \lim_{n_c \to \infty} -2 \times \ln(P_T) \times \underbrace{{P_T}^{\frac{2}{n_c}}}_{1} \\
&= -2 \times \ln(P_T)
\end{aligned} \tag{C.8}$$

The expression in Eq. (C.8) provides the limit of $f(n_c)$ when n_c approaches infinity for the case $i = k$. Next, we provide the expression for obtaining the limit of $f(n_c)$ when n_c approaches infinity for the case $i < k$.

For the second case, *i.e.* when $i < k$, the limit of $f(n_c)$ when n_c approaches infinity is evaluated as:

$$\lim_{n_c \to \infty} f(n_c) = \lim_{n_c \to \infty} n_c \times (1 - {P_T}^{\frac{2}{n_c}})^b \tag{C.9}$$

Observe that the parameter b in Eq. (C.9) is strictly larger than one. By directly replacing n_c with infinity in Eq. (C.9) we get that the limit is undefined as we need to multiply infinity with zero. However, this allows us to use the l'Hôpital's rule, in which case we get:

$$\begin{aligned}
\lim_{n_c\to\infty} f(n_c) &= \lim_{n_c\to\infty} n_c \times (1 - {P_T}^{\frac{2}{n_c}})^b \\
&= \lim_{n_c\to\infty} \frac{(1 - {P_T}^{\frac{2}{n_c}})^b}{\frac{1}{n_c}} \\
&= \lim_{n_c\to\infty} \frac{b \times (1 - {P_T}^{\frac{2}{n_c}})^{b-1} \times (-\ln(P_T)) \times {P_T}^{\frac{2}{n_c}} \times \frac{-2}{{n_c}^2}}{\frac{-1}{{n_c}^2}} \\
&= \lim_{n_c\to\infty} b \times \underbrace{(1 - {P_T}^{\frac{2}{n_c}})^{b-1}}_{0} \times 2 \times \ln(P_T) \times \underbrace{{P_T}^{\frac{2}{n_c}}}_{1} \\
&= 0 \qquad \text{(C.10)}
\end{aligned}$$

Using Eq. (C.8) and Eq. (C.10), and replacing them in Eq. (C.5) we get:

$$\begin{aligned}
\lim_{n_c\to\infty} p_{n_c}(k) &= \frac{{P_T}^2}{k!} \times \sum_{i=1}^{k} a_i \left(\lim_{n_c\to\infty} f(n_c) \right)^i \\
&= \frac{{P_T}^2}{k!} \times \sum_{i=1}^{k-1} a_i \underbrace{\left(\lim_{n_c\to\infty} f(n_c) \right)^i}_{\text{using Eq. (C.10)}} + \frac{{P_T}^2}{k!} \times a_k \underbrace{\left(\lim_{n_c\to\infty} f(n_c) \right)^k}_{\text{using Eq. (C.8)}} \\
&= 0 + \frac{{P_T}^2}{k!} \times \underbrace{a_k}_{1} (-2 \times \ln(P_T))^k \\
&= \frac{{P_T}^2}{k!} \times (-2 \times \ln(P_T))^k \qquad \text{(C.11)}
\end{aligned}$$

Using Eq. (C.2) and Eq. (C.11), we evaluate the limit $\bar{\lambda}_{n_c}(k)$ as:

$$\bar{\lambda}_{n_c}(k) = {P_T}^2 \sum_{i=0}^{k} \frac{(-2 \times \ln(P_T))^i}{i!} \qquad \text{(C.12)}$$

Observe that the expression given in Eq. (C.12) is the same expression as earlier stated in **Theorem 4**.

To verify that the limit is correct, we show that the fundamental property of the LoC (see Eq. (C.1)) holds when n_c approaches infinity, *i.e.*:

$$\begin{aligned}
\lim_{k\to\infty} \bar{\lambda}_{n_c}(k) &= \lim_{k\to\infty} {P_T}^2 \sum_{i=0}^{k} \frac{(-2 \times \ln(P_T))^i}{i!} \\
&= {P_T}^2 \times \lim_{k\to\infty} \sum_{i=0}^{k} \frac{i!}{(-2 \times \ln(P_T))^i} \\
&= {P_T}^2 \times \sum_{i=0}^{\infty} \frac{(-2 \times \ln(P_T))^i}{i!} \qquad \text{(C.13)}
\end{aligned}$$

Observe that the right-hand side of Eq. (C.13) includes an infinite series of the type:

$$\sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{C.14}$$

The infinite series presented in Eq. (C.14) is evaluated with following closed-form expression:

$$\sum_{i=0}^{\infty} \frac{x^i}{i!} = e^x \tag{C.15}$$

Using Eq. (C.15), we re-write the expression in Eq. (C.13) with:

$$\begin{aligned} \lim_{k\to\infty} \bar{\lambda}_{n_c}(k) &= {P_T}^2 \times \sum_{i=0}^{\infty} \frac{\left(-2 \times \ln(P_T)\right)^i}{i!} \\ &= {P_T}^2 \times e^{(-2\times\ln(P_T))} \\ &= {P_T}^2 \times e^{\ln(\frac{1}{{P_T}^2})} \\ &= {P_T}^2 \times \frac{1}{{P_T}^2} \\ &= 1 \end{aligned} \tag{C.16}$$

By this, we show that the fundamental property of the LoC holds also when n_c approaches infinity. We showed it while using the expression for evaluating $\bar{\lambda}_{n_c}(k)$. This verifies the correctness of the expression presented with Eq. (C.12).

■

Appendix D

In Section 5.2, we presented an expression to obtain ${}_k n_c^*$, which for a given $k > 0$ provides the minimal completion time with respect to k. The expression was given with the following equation:

$$ {}_k n_c^* = \sqrt{k \times \frac{T}{\tau}} \tag{D.1} $$

The problem with the Eq. (D.1) is that its right-hand side may be evaluated as a non-integer number. However, only integer values can be assigned to n_c.

In this regard, the straightforward solution would be rounding the value, evaluated with the right-hand side of Eq. (D.1), to the closest integer value. However, we show in this section that simple rounding does not always provide the correct value of ${}_k n_c^*$. Instead, it is required to compare the completion time for the values of n_c which are equal to the upper and the lower integer bound of the expression given in Eq. (D.1). Next, we show why rounding to the closest integer does not always provide the correct optimal value of n_c.

Let us assume that for a given k the expression in Eq. (D.1) is evaluated as a real number, *i.e.* ${}_k n_c^* \in \mathbb{R}$. In such case, the upper and the lower integer bound of ${}_k n_c^*$ can be expressed with the following expressions:

$$ n_{c_L} = \lfloor {}_k n_c^* \rfloor =_k n_c^* - \sigma_1 \tag{D.2} $$

$$ n_{c_U} = \lceil {}_k n_c^* \rceil =_k n_c^* + \sigma_2 \tag{D.3} $$

The terms σ_1 and σ_2 used in Eq. (D.2) and Eq. (D.3) represent real numbers that belong in the interval $(0, 1)$. Since the difference between n_{c_U} and n_{c_L} is

exactly equal to 1, the following relation exist between σ_1 and σ_2:

$$\begin{aligned} n_{c_U} - n_{c_L} &= 1 \\ {}_k n_c^* + \sigma_2 - {}_k n_c^* + \sigma_1 &= 1 \\ \sigma_2 + \sigma_1 &= 1 \end{aligned} \tag{D.4}$$

The following three cases exist:

- *Case I:* $\sigma_2 < \sigma_1$, meaning that ${}_k n_c^*$ is closer to the upper bound n_{c_U};
- *Case II:* $\sigma_2 = \sigma_1$, meaning that ${}_k n_c^*$ is exactly in the middle between n_{c_L} and n_{c_U}; and
- *Case III:* $\sigma_2 > \sigma_1$, meaning that ${}_k n_c^*$ is closer to the lower bound n_{c_L}

Next, for each case we compare the completion time obtained for n_{c_L} and n_{c_U}. We perform the comparison by evaluating the following expression:

$$\begin{aligned} & {}^{n_{c_U}} t_k - {}^{n_{c_L}} t_k \\ = \;& T + n_{c_U} \times \tau + k \times (\frac{T}{n_{c_U}} + \tau) - T - n_{c_L} \times \tau - k \times (\frac{T}{n_{c_L}} + \tau) \\ = \;& \underbrace{(n_{c_U} - n_{c_L})}_{1} \times \tau + k \times \frac{T \times \overbrace{(n_{c_L} - n_{c_U})}^{-1}}{n_{c_U} \times n_{c_L}} \\ = \;& \tau - \frac{k \times T}{n_{c_U} \times n_{c_L}} \\ = \;& \tau \times \frac{n_{c_U} \times n_{c_L} - k \times \overbrace{\frac{T}{\tau}}^{\alpha}}{n_{c_U} \times n_{c_L}} \quad \text{using Eq. (D.1), Eq. (D.2), Eq. (D.3) and } \alpha = \frac{T}{\tau} \\ = \;& \tau \times \frac{\left(\sqrt{k \times \alpha} + \sigma_2\right) \times \left(\sqrt{k \times \alpha} - \sigma_1\right) - k \times \alpha}{\left(\sqrt{k \times \alpha} + \sigma_2\right) \times \left(\sqrt{k \times \alpha} - \sigma_1\right)} \\ = \;& \tau \times \frac{k \times \alpha + (\sigma_2 - \sigma_1) \times \sqrt{k \times \alpha} - \sigma_2 \times \sigma_1 - k \times \alpha}{\left(\sqrt{k \times \alpha} + \sigma_2\right) \times \left(\sqrt{k \times \alpha} - \sigma_1\right)} \\ = \;& \tau \times \frac{k \times \alpha + (\sigma_2 - \sigma_1) \times \sqrt{k \times \alpha} - \sigma_2 \times \sigma_1 - k \times \alpha}{\left(\sqrt{k \times \alpha} + \sigma_2\right) \times \left(\sqrt{k \times \alpha} - \sigma_1\right)} \\ = \;& \tau \times \frac{(\sigma_2 - \sigma_1) \times \sqrt{k \times \alpha} - \sigma_2 \times \sigma_1}{\left(\sqrt{k \times \alpha} + \sigma_2\right) \times \left(\sqrt{k \times \alpha} - \sigma_1\right)} \end{aligned}$$

$$= \frac{\tau}{\left(\sqrt{k \times \alpha} + \sigma_2\right) \times \left(\sqrt{k \times \alpha} - \sigma_1\right)} \times \left((\sigma_2 - \sigma_1) \times \sqrt{k \times \alpha} - \sigma_2 \times \sigma_1\right) \tag{D.5}$$

Observe that Eq. (D.5) can result in either a positive or a negative value. Since τ is always positive, and the expression $\left(\sqrt{k \times \alpha} + \sigma_2\right) \times \left(\sqrt{k \times \alpha} - \sigma_1\right)$ is strictly positive (it multiplies two integer values larger than one), then the sign of Eq. (D.5) depends only on the following term:

$$(\sigma_2 - \sigma_1) \times \sqrt{k \times \alpha} - \sigma_2 \times \sigma_1 \tag{D.6}$$

If Eq. (D.6) results with a negative value, then the minimal completion time with respect to the given k is obtained for ${}_k n_c^* = n_{c_L}$. Otherwise, if Eq. (D.6) results with a positive value, or zero, the minimal completion time with respect to the given k is obtained for ${}_k n_c^* = n_{c_U}$. Observe that if Eq. (D.5) results with zero, it means that the same completion time will be achieved for both n_{c_L} and n_{c_U}. However, choosing n_{c_U} is more beneficial as for a larger value of n_c, $\lambda_{n_c}(k)$ is higher (see **Theorem 3**).

For *Case I*, the relation between σ_1 and σ_2 is as follows:

$$\sigma_2 < \sigma_1 \tag{D.7}$$

Next, we show that when Eq. (D.7) holds, Eq. (D.6) results with a negative value. Since both σ_1 and σ_2 are positive, the term $\sigma_1 \times \sigma_2$ is also positive. This justifies the following inequality:

$$(\sigma_2 - \sigma_1) \times \sqrt{k \times \alpha} - \sigma_2 \times \sigma_1 < (\sigma_2 - \sigma_1) \times \sqrt{k \times \alpha} \tag{D.8}$$

When Eq. (D.7) holds, the right-hand side of Eq. (D.8) will be strictly lower than zero, *i.e.* $\sigma_2 - \sigma_1 < 0$. Thus, we get:

$$(\sigma_2 - \sigma_1) \times \sqrt{k \times \alpha} < 0 \tag{D.9}$$

Using Eq. (D.9) in Eq. (D.8) leads to the following inequality:

$$(\sigma_2 - \sigma_1) \times \sqrt{k \times \alpha} - \sigma_2 \times \sigma_1 < 0 \tag{D.10}$$

Observe that the left-hand side of Eq. (D.10) represents the same expression as Eq. (D.6). Thus, Eq. (D.10) shows that when Eq. (D.7) holds, Eq. (D.6) results with a negative value. In other words, when ${}_k n_c^*$, computed with Eq. (D.1) is closer to the upper bound, n_{c_U} (Eq. (D.3)), the minimal completion time with respect to k is obtained for n_{c_U}. This suggests that simple integer rounding would provide the correct optimal value of n_c for the given k.

For *Case II*, the relation between σ_1 and σ_2 is as follows:

$$\sigma_2 = \sigma_1 \tag{D.11}$$

Eq. (D.11) reduces Eq. (D.6) to the following term:

$$-\sigma_2 \times \sigma_1 \tag{D.12}$$

Since both σ_2 and σ_1 are positive, the product term $\sigma_2 \times \sigma_1$ is also positive. Thus, the expression in Eq. (D.12) is evaluated as negative. By this we show that when Eq. (D.11) holds Eq. (D.6) results with a negative value. Hence, when ${}_k n_c^*$, computed with Eq. (D.1) is exactly in the middle between n_{c_L} (Eq. (D.2)) and n_{c_U} (Eq. (D.3)), the minimal completion time with respect to k is obtained for n_{c_U}. Observe that integer rounding with a tie-breaking rule to always round half up would provide the correct optimal value of n_c for the given k.

For *Case III*, the relation between σ_1 and σ_2 is as follows:

$$\sigma_2 > \sigma_1 \tag{D.13}$$

When Eq. (D.13) holds, it is not straightforward to deduct whether Eq. (D.6) results in a positive or a negative value, as it was for *Case I* and *Case II*.

Using Eq. (D.4), we substitute σ_2 with the following expression:

$$\sigma_2 = 1 - \sigma_1 \tag{D.14}$$

Replacing σ_2, Eq. (D.14), in Eq. (D.13) leads to the fact that Eq. (D.13) holds only for $\sigma_1 < 0.5$. Using Eq. (D.14) transforms Eq. (D.6) into:

$$\begin{aligned} & (\sigma_2 - \sigma_1) \times \sqrt{k \times \alpha} - \sigma_2 \times \sigma_1 \\ = \;& (1 - 2 \times \sigma_1) \times \sqrt{k \times \alpha} - (1 - \sigma_1) \times \sigma_1 \\ = \;& \sqrt{k \times \alpha} - 2 \times \sqrt{k \times \alpha} \times \sigma_1 - \sigma_1 + {\sigma_1}^2 \\ = \;& {\sigma_1}^2 - \left(2 \times \sqrt{k \times \alpha} + 1\right) \times \sigma_1 + \sqrt{k \times \alpha} \end{aligned} \tag{D.15}$$

The expression in Eq. (D.15) represents a quadratic function of σ_1 that reaches a minimum (the coefficient in front of the term ${\sigma_1}^2$ is positive). Note that the constant α used in Eq. (D.15) represents the ratio of the processing time T and the checkpointing overhead τ, *i.e.* $\alpha = \frac{T}{\tau}$. The roots of Eq. (D.15) are evaluated as:

$$\sigma_{1_1} = \frac{1}{2} - \frac{\sqrt{4 \times k \times \alpha + 1} - 2 \times \sqrt{k \times \alpha}}{2} \tag{D.16}$$

$$\sigma_{1_2} = \frac{1}{2} + \frac{2 \times \sqrt{k \times \alpha} + \sqrt{4 \times k \times \alpha + 1}}{2} \tag{D.17}$$

Since Eq. (D.15) reaches a minimum, and it has two different roots σ_{1_1} and σ_{1_2}, this leads to the fact that Eq. (D.15) results with a positive value for any σ_1 that belongs to$(-\infty, \sigma_{1_1}) \cup (\sigma_{1_2}, \infty)$, and Eq. (D.15) results with a negative value for $\sigma_1 \in (\sigma_{1_1}, \sigma_{1_2})$.

By observing Eq. (D.17) we conclude that σ_{1_2} is strictly larger than 0.5 (a positive term is added to the term $\frac{1}{2}$). On the other hand, by observing Eq. (D.16) we conclude that σ_{1_1} is strictly lower than 0.5. However, σ_{1_1} is larger than zero. This can be shown by re-writing Eq. (D.16) with the following expression:

$$\begin{aligned}\sigma_{1_1} =& \frac{1}{2} - \frac{1}{2} \times \frac{\sqrt{4 \times k \times \alpha + 1} + 2 \times \sqrt{k \times \alpha}}{\sqrt{4 \times k \times \alpha + 1} + 2 \times \sqrt{k \times \alpha}} \left(\sqrt{4 \times k \times \alpha + 1} - 2 \times \sqrt{k \times \alpha} \right) \\ =& \frac{1}{2} - \frac{1}{2} \times \frac{4 \times k \times \alpha + 1 - 4 \times k \times \alpha}{\sqrt{4 \times k \times \alpha + 1} + 2 \times \sqrt{k \times \alpha}} \\ =& \frac{1}{2} - \frac{1}{2} \times \frac{1}{\sqrt{4 \times k \times \alpha + 1} + 2 \times \sqrt{k \times \alpha}} \end{aligned} \tag{D.18}$$

Observe the second term in Eq. (D.18). It is obtained by multiplying the constant $\frac{1}{2}$ with the following expression:

$$\frac{1}{\sqrt{4 \times k \times \alpha + 1} + 2 \times \sqrt{k \times \alpha}} \tag{D.19}$$

The expression in Eq. (D.19) is a positive term, and it is lower than one. This comes from the fact that k is an integer greater than zero, and the constant α only has meaning in the context of RRC if it is larger than one. Observe that if α is equal to one, that means that the checkpointing overhead of taking a single checkpoint takes the same amount of time as re-executing the entire job. Thus, even if we assume the unlikely scenario of $\alpha = 1$ and the lowest integer greater than zero, *i.e.* $k = 1$, Eq. (D.19) results in a value which is lower than one, *i.e.* $1/(2 + \sqrt{5})$. This ensures that the second term in Eq. (D.18) will be lower than $\frac{1}{2}$, which in turns ensures that σ_{1_1} will be larger than zero.

As *Case III* is applicable only when $\sigma_1 \in [0, 0.5)$, we conclude that σ_{1_1} splits the interval $[0, 0.5)$ into two disjoint intervals $[0, \sigma_{1_1})$ and $[\sigma_{1_1}, 0.5)$. When $\sigma_1 \in [0, \sigma_{1_1})$ Eq. (D.6) results with a positive value. It means that the minimal completion time with respect to k is obtained for n_{c_L}. Observe that since $\sigma_1 < 0.5$, n_{c_L} is the closest integer value to ${}_k n_c^*$ computed with Eq. (D.1). Thus, applying integer rounding would select n_{c_L}. However, when $\sigma_1 \in (\sigma_{1_1}, 0.5)$ Eq. (D.6) results with a negative value or zero. It means that the minimal completion time with respect to k is obtained for n_{c_U}, even though n_{c_U} is not the closest integer to ${}_k n_c^*$ computed with Eq. (D.1) (observe that σ_1 is still lower than 0.5). This shows that simple integer rounding cannot be used to obtain the optimal value of n_c for the given k.

References

[1] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," *Micro, IEEE*, vol. 23, no. 4, pp. 14–19, 2003.

[2] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai, "Impact of cmos process scaling and soi on the soft error rates of logic processes," in *VLSI Technology, 2001. Digest of Technical Papers. 2001 Symposium on*, pp. 73–74, IEEE, 2001.

[3] R. Baumann, "The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction," in *Electron Devices Meeting, 2002. IEDM'02. International*, pp. 329–332, IEEE, 2002.

[4] V. Chandra and R. Aitken, "Impact of technology and voltage scaling on the soft error susceptibility in nanoscale cmos," in *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*, pp. 114–122, IEEE, 2008.

[5] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pp. 389–398, IEEE, 2002.

[6] N. Miskov-Zivanov and D. Marculescu, "Multiple transient faults in combinational and sequential circuits: A systematic approach," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 29, pp. 1614–1627, Oct 2010.

[7] I. Koren and C. Krishna, *Fault-tolerant systems*. Morgan Kaufmann, 2007.

[8] G. Saggese, N. Wang, Z. Kalbarczyk, S. Patel, and R. Iyer, "An experimental study of soft errors in microprocessors," *Micro, IEEE*, vol. 25, pp. 30–39, Nov 2005.

[9] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, pp. 305–316, Sept 2005.

[10] R. Baumann, "Soft errors in advanced semiconductor devices-part i: the three radiation sources," *Device and Materials Reliability, IEEE Transactions on*, vol. 1, pp. 17–22, Mar 2001.

[11] J. Borel, "European design automation roadmap," *March*, 2009.

[12] T. Juhnke and H. Klar, "Calculation of the soft error rate of submicron cmos logic circuits," *Solid-State Circuits, IEEE Journal of*, vol. 30, no. 7, pp. 830–834, 1995.

[13] M. Mishra and S. Goldstein, "Defect tolerance at the end of the roadmap," *Nano, quantum and molecular computing*, pp. 73–108, 2004.

[14] N. Sirisantana, B. Paul, and K. Roy, "Enhancing yield at the end of the technology roadmap," *Design & Test of Computers, IEEE*, vol. 21, no. 6, pp. 563–571, 2004.

[15] S. Borkar, "Design challenges of technology scaling," *Micro, IEEE*, vol. 19, no. 4, pp. 23–29, 1999.

[16] S. Mukherjee, *Architecture design for soft errors*. Morgan Kaufmann, 2008.

[17] D. Blaauw, S. Kalaiselvan, K. Lai, W. Ma, S. Pant, C. Tokunaga, S. Das, and D. Bull, "Razor ii: In situ error detection and correction for pvt and ser tolerance," in *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pp. 400–622, IEEE, 2008.

[18] A. Avizienis, "Fault-tolerance: The survival attribute of digital systems," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1109–1125, 1978.

[19] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, pp. 11–33, Jan 2004.

[20] J.-C. Laprie, "Dependable computing and fault tolerance : Concepts and terminology," in *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*, pp. 2–, Jun 1995.

[21] J. Laprie, "Dependable computing and fault-tolerance," *Digest of Papers FTCS-15*, pp. 2–11, 1985.

[22] A. Avizienis and J. Kelly, "Fault tolerance by design diversity- concepts and experiments," *Computer*, vol. 17, pp. 67–80, 1984.

[23] J. Sosnowski, "Transient fault tolerance in digital systems," *Micro, IEEE*, vol. 14, no. 1, pp. 24–35, 1994.

[24] R. Al-Omari, A. Somani, and G. Manimaran, "A new fault-tolerant technique for improving schedulability in multiprocessor real-time systems," in *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pp. 8–pp, IEEE, 2001.

[25] D. Pradhan and N. Vaidya, "Roll-forward checkpointing scheme: A novel fault-tolerant architecture," *Computers, IEEE Transactions on*, vol. 43, no. 10, pp. 1163–1174, 1994.

[26] J. Bruno and E. Coffman Jr, "Optimal fault-tolerant computing on multiprocessor systems," *Acta Informatica*, vol. 34, no. 12, pp. 881–904, 1997.

[27] A. Avizienis, "Arithmetic error codes: Cost and effectiveness studies for application in digital system design," *Computers, IEEE Transactions on*, vol. C-20, pp. 1322–1331, Nov 1971.

[28] T. R. Rao, "Biresidue error-correcting codes for computer arithmetic," *Computers, IEEE Transactions on*, vol. C-19, pp. 398–402, May 1970.

[29] W. C. Huffman and V. Pless, *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003. Cambridge Books Online.

[30] J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata studies*, vol. 34, pp. 43–98, 1956.

[31] K. Alstrom and J. Torin, "Future architecture for flight control systems," in *Digital Avionics Systems, 2001. DASC. 20th Conference*, vol. 1, pp. 1B5–1, IEEE, 2001.

[32] R. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[33] J. Wakerly, "Microcomputer reliability improvement using triple-modular redundancy," *Proceedings of the IEEE*, vol. 64, no. 6, pp. 889–895, 1976.

[34] F. Kastensmidt, L. Sterpone, L. Carro, and M. Reorda, "On the optimal design of triple modular redundancy logic for sram-based fpgas," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 2*, pp. 1290–1295, IEEE Computer Society, 2005.

[35] R. Hentschke, F. Marques, F. Lima, L. Carro, A. Susin, and R. Reis, "Analyzing area and performance penalty of protecting different digital modules with hamming code and triple modular redundancy," in *Integrated Circuits and Systems Design, 2002. Proceedings. 15th Symposium on*, pp. 95–100, IEEE, 2002.

[36] R. Kumar, K. Izui, M. Yoshimura, and S. Nishiwaki, "Optimal modular redundancy using hierarchical genetic algorithms," in *Reliability and Maintainability Symposium, 2007. RAMS '07. Annual*, pp. 398–404, Jan 2007.

[37] K. Heidtmann, "Deterministic reliability-modeling of dynamic redundancy," *Reliability, IEEE Transactions on*, vol. 41, pp. 378–385, Sep 1992.

[38] I. Koren and E. Shalev, "Reliability analysis of hybrid redundancy systems," *IEEE Proceedings on Computer and Digital Techniques*, vol. 4, pp. 31–36, Jan 1984.

[39] P. T. de Sousa and F. P. Mathur, "Sift-out modular redundancy," *Computers, IEEE Transactions on*, vol. C-27, pp. 624–627, July 1978.

[40] P. L'Ecuyer and J. Malenfant, "Computing optimal checkpointing strategies for rollback and recovery systems," *Computers, IEEE Transactions on*, vol. 37, no. 4, pp. 491–496, 1988.

[41] E. Gelenbe and M. Hernández, "Optimum checkpoints with age dependent failures," *Acta Informatica*, vol. 27, no. 6, pp. 519–531, 1990.

[42] C. Krishna, Y. Lee, and K. Shin, "Optimization criteria for checkpoint placement," *Communications of the ACM*, vol. 27, no. 10, pp. 1008–1012, 1984.

[43] V. Nicola, *Checkpointing and the modeling of program execution time*. University of Twente, Department of Computer Science and Department of Electrical Engineering, 1994.

[44] E. Coffman Jr and E. Gilbert, "Optimal strategies for scheduling checkpoints and preventive maintenance," *Reliability, IEEE Transactions on*, vol. 39, no. 1, pp. 9–18, 1990.

[45] Y. Ling, J. Mi, and X. Lin, "A variational calculus approach to optimal checkpoint placement," *Computers, IEEE Transactions on*, vol. 50, no. 7, pp. 699–708, 2001.

[46] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processors-a survey," *Computers, IEEE Transactions on*, vol. 37, no. 2, pp. 160–174, 1988.

[47] A. Ziv and J. Bruck, "Analysis of checkpointing schemes with task duplication," *Computers, IEEE Transactions on*, vol. 47, no. 2, pp. 222–227, 1998.

[48] J. Smolens, B. Gold, J. Kim, B. Falsafi, J. Hoe, and A. Nowatryk, "Fingerprinting: bounding soft-error-detection latency and bandwidth," *Micro, IEEE*, vol. 24, pp. 22–29, Nov 2004.

[49] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Păun, and S. Scott, "A reliability-aware approach for an optimal checkpoint/restart model in hpc environments," in *Cluster Computing, 2007 IEEE International Conference on*, pp. 452–457, 2007.

[50] K. Shin, T. Lin, and Y. Lee, "Optimal checkpointing of real-time tasks," *Computers, IEEE Transactions on*, vol. 100, no. 11, pp. 1328–1341, 1987.

[51] S. Kwak, B. Choi, and B. Kim, "An optimal checkpointing-strategy for real-time control systems under transient faults," *Reliability, IEEE Transactions on*, vol. 50, no. 3, pp. 293–301, 2001.

[52] N. Naksinehaboon, Y. Liu, C. Leangsuksun, R. Nassar, M. Păun, and S. Scott, "Reliability-aware approach: An incremental checkpoint/restart model in hpc environments," in *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on*, pp. 783–788, 2008.

[53] J. Sancho, F. Petrini, G. Johnson, and E. Frachtenberg, "On the feasibility of incremental checkpointing for scientific computing," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pp. 58–, 2004.

[54] R. Gioiosa, J. Sancho, S. Jiang, and F. Petrini, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pp. 9–9, 2005.

[55] S.-M. Ryu and D.-J. Park, "Checkpointing for the reliability of real-time systems with on-line fault detection," in *Embedded and Ubiquitous Computing - EUC 2005* (L. Yang, M. Amamiya, Z. Liu, M. Guo, and F. Rammig, eds.), vol. 3824 of *Lecture Notes in Computer Science*, pp. 194–202, Springer Berlin Heidelberg, 2005.

[56] Y. Zhang and K. Chakrabarty, "Fault recovery based on checkpointing for hard real-time embedded systems," in *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, pp. 320–327, IEEE, 2003.

[57] M. Väyrynen, V. Singh, and E. Larsson, "Fault-tolerant average execution time optimization for general-purpose multi-processor system-on-chips," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 484–489, European Design and Automation Association, 2009.

[58] S. Hiroyama, T. Dohi, and H. Okamura, "Comparison of aperiodic checkpoint placement algorithms," in *Advanced Computer Science and Information Technology* (G. Tomar, R.-S. Chang, O. Gervasi, T.-h. Kim, and S. Bandyopadhyay, eds.), vol. 74 of *Communications in Computer and Information Science*, pp. 145–156, Springer Berlin Heidelberg, 2010.

[59] S. Hiroyama, T. Dohi, and H. Okamura, "Aperiodic checkpoint placement algorithms - survey and comparison," *Journal of Software Engineering and Applications*, vol. 6, pp. 41–53, 2013.

[60] T. Ozaki, T. Dohi, and N. Kaio, "Numerical computation algorithms for sequential checkpoint placement," *Perform. Eval.*, vol. 66, pp. 311–326, June 2009.

[61] A. Ziv and J. Bruck, "An on-line algorithm for checkpoint placement," *Computers, IEEE Transactions on*, vol. 46, no. 9, pp. 976–985, 1997.

[62] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Norwell, MA, USA: Kluwer Academic Publishers, 1st ed., 1997.

[63] J. Liu, *Real-time systems*. Prentice Hall PTR, 2000.

[64] C. Aubrun, D. Simon, and Y.-Q. Song, *Co-design Approaches for Dependable Networked Control Systems*. ISTE Wiley, Jan. 2010.

[65] A. TeÅąanoviÄĞ, M. Amirijoo, and J. Hansson, "Providing configurable qos management in real-time systems with qos aspect packages," in *Transactions on Aspect-Oriented Software Development II* (A. Rashid and M. Aksit, eds.), vol. 4242 of *Lecture Notes in Computer Science*, pp. 256–288, Springer Berlin Heidelberg, 2006.

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

[67] S. Punnekkat, A. Burns, and R. Davis, "Analysis of checkpointing for real-time systems," *Real-Time Systems*, vol. 20, no. 1, pp. 83–102, 2001.

[68] N. Kandasamy, J. Hayes, and B. Murray, "Transparent recovery from intermittent faults in time-triggered distributed systems," *Computers, IEEE Transactions on*, vol. 52, no. 2, pp. 113–125, 2003.

[69] P. Pop, V. Izosimov, P. Eles, and Z. Peng, "Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, no. 3, pp. 389–402, 2009.

[70] A. Bertossi and L. Mancini, "Scheduling algorithms for fault-tolerance in hard-real-time systems," *Real-Time Systems*, vol. 7, no. 3, pp. 229–245, 1994.

[71] A. Bertossi, A. Fusiello, and L. Mancini, "Fault-tolerant deadline-monotonic algorithm for scheduling hard-real-time tasks," in *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pp. 133–138, IEEE, 1997.

[72] A. Burns, R. Davis, and S. Punnekkat, "Feasibility analysis of fault-tolerant real-time task sets," in *Real-Time Systems, 1996., Proceedings of the Eighth Euromicro Workshop on*, pp. 29–33, IEEE, 1996.

[73] C. Han, K. Shin, and J. Wu, "A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults," *Computers, IEEE Transactions on*, vol. 52, no. 3, pp. 362–372, 2003.

[74] Y. Zhang and K. Chakrabarty, "A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 1, pp. 111–125, 2006.

[75] S. Nakagawa, S. Fukumoto, and N. Ishii, "Optimal checkpointing intervals of three error detection schemes by a double modular redundancy," *Mathematical and computer modelling*, vol. 38, no. 11, pp. 1357–1363, 2003.

[76] V. Grassi, L. Donatiello, and S. Tucci, "On the optimal checkpointing of critical tasks and transaction-oriented systems," *Software Engineering, IEEE Transactions on*, vol. 18, no. 1, pp. 72–77, 1992.

[77] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Scheduling of fault-tolerant embedded systems with soft and hard timing constraints," in *Proceedings of the conference on Design, automation and test in Europe*, pp. 915–920, ACM, 2008.

[78] S. W. Kwak and J.-M. Yang, "Probabilistic optimisation of checkpoint intervals for real-time multi-tasks," *International Journal of Systems Science*, vol. 44, no. 4, pp. 595–603, 2013.

[79] J. M. Yang and S. Kwak, "A checkpoint scheme with task duplication considering transient and permanent faults," in *Industrial Engineering and Engineering Management (IEEM), 2010 IEEE International Conference on*, pp. 606–610, Dec 2010.