7

Checkpointing and the Modeling of Program Execution Time

VICTOR F. NICOLA
University of Twente, Netherlands

ABSTRACT

Checkpointing is a commonly used technique for reducing the execution time of long-running programs in the presence of failures. With checkpointing, the status of the program under execution is saved intermittently. Upon the occurrence of a failure, the program execution is restarted from the most recent checkpoint rather than from the beginning. Due to its overhead, checkpointing may not always be beneficial, and, if it is, an optimal checkpointing strategy may be determined so as to minimize the expected execution time or to maximize the probability of the execution time not exceeding a critical limit. In this chapter we consider several models of checkpointing and recovery in a program in order to derive the distribution of program execution time or its expectation. We make important extensions to some existing results as well as introduce and analyze new models. It is shown that the expected execution time increases linearly (exponentially) with the processing requirement in the presence (absence) of checkpointing. Furthermore, these models can be used to compare different checkpointing strategies and to determine an optimal interval between checkpoints. Throughout the chapter, and whenever appropriate, we attempt to relate to other work in the existing literature.

7.1 INTRODUCTION

Fault-tolerance is a much desired feature of long-running software applications that may arise in a variety of contexts, such as numerically intensive computational physics, numerical optimization problems, simulation of complex systems, complex queries in large relational databases, and others. The reliability of such applications may be compromised due to exter-

Software Fault Tolerance, Edited by Michael R. Lyu

nal transient faults, such as power failure, disk failure, processor failure, and other types of system malfunctions, or due to permanent faults, such as those internal to the program itself. Techniques for fault-tolerance (see, for example, [And81] and Chapters 1 to 4 of this book) can be used to enhance the reliability of such applications, however, often at the expense of a performance degradation and/or additional resources. For example, adding a restart capability after repair (or replacement) will guarantee the successful completion of the program in the presence of failures [Ran75, Puc92]. However, such a restart capability does not guard against excessive run times due to repeated failures followed by prohibitive reprocessing. The execution time of a program is defined as the time to complete computation, including repairs and restarts after failures. This also includes additional overhead required by the fault-tolerance procedure, if any. The execution time is an important metric which is commonly used to evaluate and compare the performance of different fault-tolerance procedures in software applications. It is well known (see, for example, [Dud83, Nic86]) that, in the presence of random failures (and a restart capability), the expected execution time of a fault-tolerant software (or a program) typically grows exponentially with its processing requirement. This is due to failures which cause the program to restart from its beginning (thus losing all useful processing done thus far.) Checkpointing is a commonly used technique in fault-tolerant software applications and long-running programs to enhance their reliability and to increase their computational efficiency (see, e.g., [Cha72a, Cha72b, You74].) Checkpointing consists of intermittently saving the current state of program execution in a reliable storage, so that when a (transient) failure occurs, [1] the program execution may restart from the most recent checkpoint, rather than from its beginning (thus losing only a part of the useful processing done thus far.)

With checkpointing, the expected execution time of a program typically grows only linearly with its processing requirement (see, for example, [Dud83, Kul90].) After a failure, the process of reloading the program status saved at the most recent checkpoint is often called a *rollback*. (Note that the rollback time may be conveniently added to the repair time for the purpose of analysis.) The reprocessing of the program, starting from the most recent checkpoint and until the point just before the failure, is often called a *recovery* process.

Clearly, more checkpoints means less reprocessing after failures. If checkpoints were costless, then the execution time would reduce monotonically with the number of checkpoints in the program. However, usually there is checkpointing overhead and, therefore, a trade-off exists. In other words, there is usually an optimal number (or frequency) of checkpoints which optimizes a certain performance measure. For example, in a production environment, a likely objective would be to minimize the expected execution time. In a real-time environment, the likely objective is to maximize the probability of the execution time not exceeding a given hard deadline.

Over the years, there has been a huge body of literature covering a wide range of issues related to checkpointing and recovery in a variety of systems. These issues range from architectural and implementation aspects to functional and performance considerations. There

[1] We assume that a failure is detected as soon as it occurs. In practice, a latency may exist between the failure occurrence and its detection [She75]. Failure detection requires the use of a continuous error-checking mechanism, however, checkpointing can also be used for the detection of latent failures [Pra89]. A latent failure may not be detected during program execution, which could result in an erroneous computation [Shi87]. Otherwise, a latency between failure occurrence and detection may cause the program to restart from an earlier checkpoint or from the beginning. In [Kor86], a model is considered in which one or more recovery procedures (namely, instruction retries, program rollbacks to earlier checkpoints and program restarts) may be used depending on the length of the detection latency.

is no way to cover all these aspects in a single chapter or even a single book. In fact, in the domain of performance related issues and over the past two decades, there has been a great number of papers, too many to list. Most of these papers deal with the modeling and analysis of various checkpointing strategies at the system level (as opposed to that at the program level.) It should be mentioned, however, that there are many issues that are common to checkpointing and rollback recovery at the system and the program levels. As an example of system level checkpointing, one may consider a transaction-oriented database system [Hae83, Reu84, Ver78]. In such a system, checkpoints are inserted between transactions according to some strategy, with the objective of enhancing the overall reliability/integrity of the database as well as improving an overall system performance measure, such as the system availability (see [Cha75a, Cha75b, DeS90, Don88, Gel79, Gel90, Lec88, Loh77, Mik79, Nic83a, Nic83b, Shn73, Tan84] and others) or the mean response time of a transaction (see [Bac81a, Bac81b, Dud84, Gel78, Kul90, Nic83a, Nic83b, Nic86, Nic90b] and others.) Far less is reported on checkpointing at the program level and its effects on program performance (see [Bog92, Bro79, Cof88, Dud83, Gei88, Gra92, Kul90, Kri84, Leu84, Shi87, Tou84, Upa86, Upa88] and others.)

The main focus in this chapter is the modeling of checkpointing at the program level and the analysis of the program execution time. Therefore, in Section 7.2 we will briefly review some of the work relevant to this particular focus. However, we would like to direct the reader's attention to the closely related work at the system level (some references are included in this chapter.) We also consider some useful models of program performance in the presence of failures, without and with checkpointing. In Section 7.3, we describe the basic model and consider program execution in the presence of failures and without checkpointing. Then, under common basic assumptions, we analyze the program execution time with different checkpointing strategies. In the process, we extend previous models and introduce new ones. In Section 7.4, we consider a model with fixed (deterministic) productive time between checkpoints. Models with generally distributed productive time between checkpoints are considered in Section 7.5. A model with a random checkpointing strategy is analyzed in Section 7.6. In Section 7.7, we conclude with a summary and possible directions for further research.

7.2 RELATED WORK

In this section we give a brief overview of existing literature related to the modeling and performance analysis of checkpointing and recovery strategies at the program level. This literature coverage is neither exhaustive nor complete, but it is intended to provide an adequate background to help the reader place the present treatment of the subject in its proper context.

In most strategies, checkpoints are scheduled with respect to the productive time (or the progress of computation), i.e., excluding the time spent in repair and reprocessing after failures. [2] In particular, because of their established optimality (under fairly general assumptions) with respect to some commonly used criteria, strategies with a deterministic productive time between checkpoints have received a considerable attention from analysts and practitioners (see, e.g., [Bro79, Dud83, Gei88, Gra92].) However, in practice, it is not always possible to place checkpoints equally spaced in a program, as this depends to a large extent on the structure of the program (see, e.g., [Cha72b, Lon92].) Furthermore, for certain criteria and/or

[2] For a more complete discussion on checkpoint insertion based on the progress of computation in a program, the reader is referred to [Lon92].

accurate assumptions about the system, the optimal checkpointing strategy is not necessarily equidistant. For example, in [Tan84], it is demonstrated that, for a Weibull failure distribution, "equicost" checkpointing achieves a higher system availability than an equidistant strategy. In an "equicost" strategy, a checkpoint is performed whenever the expected reprocessing cost is equal to the checkpointing cost. For Poisson failures, this strategy results in equidistant checkpoints. However, for (more typical) increasing failure rate distributions, the frequency of checkpoints increases with the time since the last failure. Another example is in [Shi87], where the possibility of incorrect program execution due to latent errors is considered. The objective is to minimize the expected execution time, subject to the constraint of keeping the probability of correct execution higher than a specified level. An optimal strategy suggests that checkpointing should be performed more frequently towards the end of the program. Such a strategy increases the probability of detecting latent errors before the end of the program, while minimizing the expected execution time. Therefore, it is also of interest to consider checkpointing strategies other than the equidistant. In particular, a few authors have considered checkpointing strategies based on a dynamic optimization approach (see, e.g., [Mag83, Tou84].) Others (as in [Nic86]) considered a non-deterministic (e.g., exponential) productive time between checkpoints to obtain the distribution of the program execution time in the presence of failures and checkpointing.

In [Bro79], a model is considered to obtain the expected execution time of a program with and without checkpointing. In [Dud83], a program is divided into equal parts and checkpoints are performed only after the execution of each program part (including repetitions due to Poisson failures.) The execution time of program parts (including the checkpointing time at the end of each part) are independent and identically distributed random variables, the sum of which is the total program execution time. In [Dud83], the distribution of the program execution time is derived and the optimal number of checkpoints which minimizes its expectation is determined. Recently, the same model is considered in [Gra92] and it is suggested that the optimal number of checkpoints be determined based on the distribution of the program execution time, rather than its expectation. In [Gei88], a system with a constraint on the repair time is considered. An expression is derived to compute the probability of completing a program, having a given processing requirement, with equally spaced checkpoints. A constraint on the operational time before program completion is also considered.

In other papers, different approaches are followed with the Poisson failures assumption relaxed and checkpoints arbitrarily placed in the program. For example, in [Leu84], an algorithm is proposed to compute the distribution of the program execution time. Then, numerical optimization is suggested to determine the optimal placement of checkpoints in the program.

For age dependent failures, the distribution of program execution time is derived in [Cof88], assuming that the system is "as-good-as-new" after each checkpoint. With this assumption, the end of each checkpoint constitutes a renewal point, thus simplifying the analysis. They show that equally spaced scheduling of checkpoints is optimal when minimizing the expected execution time or the tail of its distribution, and also when maximizing the probability of completing the program in the presence of permanent failures. Age dependent failures have also been considered in models of checkpointing in database systems (see, for example, [Gel90, Sum89, Tan84].)

In [Tou84], a stochastic dynamic programming approach is considered to determine the optimal placement of checkpoints between tasks of given (arbitrary) lengths in a program, so as to minimize the total expected program execution time. An integer programming approach is considered in [Mag83].

General models for the analysis of the program execution time have been considered, in the presence of failures, degradation and repair, but not with checkpointing (see [Gav62, Kul87, Nic86, Nic87] and others.) In these models, the occurrence of a failure may result in no loss or in a complete loss of all the work done on the program, as opposed to a partial loss of the work in the presence of program checkpoints. A similar approach is followed in [Kul90] to model and analyze the program execution time with random (Poisson) failures and checkpointing.

A random checkpointing strategy makes sense in situations when it is not possible to schedule (insert) checkpoints at the program level, or when the decision to perform a checkpoint depends on the (non-deterministic) execution environment. Consider, for instance, "dynamic" checkpoint insertion schemes [Li90], in which a real-time (system) clock is polled to decide if a checkpoint is due. Clock granularity and the workload on the system can result in different checkpoint locations for different execution runs of the same computation. Another situation is when "global" checkpointing schemes are used in concurrent processing environments (see, e.g., [Kan78, Tha86].) In such cases, checkpoint operations (on behalf of the program) may be triggered at another application level or at the system level. For example, the system may trigger a checkpoint (on behalf of some or all running application programs) whenever it is lightly loaded or in response to certain events. In situations such as the above, checkpoints are not scheduled (inserted) at the program level, and, therefore, can be viewed to occur randomly (in a way, just like failures.) With random checkpointing strategies, it is obvious that checkpoints may occur not only during useful processing, but also during reprocessing subsequent to failures. In general, the possibility of checkpointing during reprocessing is of much interest, since it guards against prohibitive reprocessing caused by repeated failures in long recovery periods. This is particularly advantageous in typical situations where the frequency of failures is high during recovery. A model that takes into account the above features of random checkpointing is considered in [Kul90].

In the remainder of this chapter, we consider some useful models of program performance in the presence of failures, without and with commonly used checkpointing strategies. In order to carry out the analysis under, more or less, the same basic assumptions, it was necessary to extend some existing models and to introduce new models which are not considered previously. From these models we derive the distribution of the program execution time and/or its expectation. This makes it possible to assess whether checkpointing is beneficial and, if so, determine the optimal number (frequency) of checkpoints in a given program.

7.3 PROGRAM EXECUTION WITHOUT CHECKPOINTING

In this section we consider the execution of a program with a given processing requirement (as measured by the computation time on a failure-free system) in the presence of failures. We assume that the program is equipped with a fault-tolerance mechanism that enables it to restart after failures. Without checkpointing, the program has to be restarted form its beginning every time a failure occurs. The execution of the program will eventually complete, when, for the first time, its entire processing requirement is completed without failures. The program execution time is defined as the time elapsed from the beginning of its execution until its completion (including repairs and reprocessing after failures.)

Let x be the processing requirement of the program, and let $T(x)$ be its execution time. (Note that the program processing requirement is identical to its execution time under failure-free environment.) Let us assume that failures occur according to a Poisson process at rate

γ. This is a commonly used and accepted assumption, particularly when failures are caused by many different and independent sources. As in most previous work, we also assume that failures are detected as soon as they occur. Following a failure, there is a repair time, denoted by the random variable R. Without loss of generality, we assume that no failures may occur during the repair time R. Clearly, due to the randomness of the failure/repair process, $T(x)$ is a random variable, even though the processing requirement x is fixed.

For a random variable, say R, we denote its probability distribution function (CDF) by $F_R(t) = P(R \leq t)$, its probability density function (pdf) by $f_R(t)$, and its Laplace-Stieltjes transform (LST) by $\phi_R(s) = E(e^{-sR}) = \int_{t=0}^{\infty} e^{-st} f_R(t)dt$. Similarly, for $T(x)$, we denote its CDF by $F_T(t,x)$, its pdf by $f_T(t,x)$, and its LST by $\phi_T(s,x)$. The following result has been established in various forms in the literature, either directly (see, e.g., [Dud83, Goy87, Gra92]) or as a special case of a more general result (see, e.g., [Kul87, Nic86].) It gives a closed form expression for the LST of the program execution time, $T(x)$, in the presence of failures. We include a proof for the purpose of illustrating the arguments, which may be used to establish other results in this chapter.

THEOREM 1. Without checkpointing, the LST of the program execution time in the presence of failures is given by

$$\phi_T(s,x) = \frac{(s+\gamma)e^{-(s+\gamma)x}}{s+\gamma(1-\phi_R(s)(1-e^{-(s+\gamma)x}))}. \tag{7.1}$$

Proof: Let H be the time to the first failure after starting program execution, then conditioning on $H = h$, we have

$$T(x)|_{H=h} = \begin{cases} x, & \text{if } \ h \geq x \\ h + R + T(x), & \text{if } \ h < x. \end{cases}$$

If $h \geq x$, then the program will complete in x units of time. If $h < x$, then a failure occurs before the completion of the program. In this case, there is a repair time R after which the program execution is restarted from its beginning, thus, again, requiring x units of uninterrupted processing time to complete. Writing the LST of $T(x)$, we have

$$\phi_T(s,x)|_{H=h} = E(e^{-sT(x)}|H=h) = \begin{cases} e^{-sx}, & \text{if } \ h \geq x \\ e^{-sh}\phi_R(s)\phi_T(s,x), & \text{if } \ h < x. \end{cases}$$

Unconditioning on H, we get

$$\begin{aligned} \phi_T(s,x) &= \int_{h=0}^{\infty} \phi_T(s,x)|_{H=h} \ \gamma e^{-\gamma h} dh \\ &= e^{-(s+\gamma)x} + \frac{\gamma\phi_R(s)\phi_T(s,x)(1-e^{-(s+\gamma)x})}{s+\gamma}, \end{aligned}$$

from which Equation 7.1 follows directly. □

The LST in Equation 7.1 can be inverted numerically or by inspection to obtain the pdf

$f_T(t,x)$. A closed form expression for $f_T(t,x)$ can be found in [Dud83]. However, in most cases, numerical inversion is necessary and can be carried out using one of many procedures for the numerical inversion of Laplace transforms (see, e.g., [DeH83, Gar88, Jag82, Kul86].) The first and higher moments of $T(x)$ can be obtained from its LST. Of particular interest is the expected program execution time, $E(T(x))$, which is given in the following corollary.

COROLLARY 1. Without checkpointing, the expected program execution time in the presence of failures is given by

$$E(T(x)) = (\frac{1}{\gamma} + E(R))(e^{\gamma x} - 1). \tag{7.2}$$

Proof: Follows directly from the LST in Equation 7.1, using the relation $E(T(x)) = \frac{-\partial \phi_T(s,x)}{\partial s}|_{s=0}$. It can also be obtained by following arguments similar to those of Theorem 1. □

Note that $E(T(x))$ grows exponentially with the processing requirement x.

Denote by $A(x)$ the expected fraction of productive time during program execution, i.e., $A(x) = x/E(T(x))$, and let A be the steady-state system availability, defined as the long run fraction of time the system is available of productive processing. Clearly, $A(x) \leq 1$ is a program (task) oriented measure, which, for sufficiently large x, approaches the system oriented measure A.

In the following sections, we consider the use of checkpointing in a program to avoid excessive execution times due to restarts in the presence of failures. We examine the effect of checkpointing on the distribution of program execution time and its expectation. In Sections 7.4 and 7.5 we analyze strategies for which the distribution of productive time between checkpoints is known; namely, deterministic and general, respectively. In Section 7.6 we consider a random checkpointing strategy.

7.4 EQUIDISTANT CHECKPOINTING

The optimality of checkpointing strategies with deterministic (fixed) productive time between checkpoints has been established at both, the system and the program, levels. For example, in [Gel79] it was shown to maximize the steady-state system availability, and in [Cof88] it was shown to minimize the expected program execution time and also the tail of its distribution. Therefore, this is an important class of checkpointing strategies, which, also because of its intuitive appeal and simplicity, has often been considered by system analysts and designers.

7.4.1 The Model

In this section, we analyze a strategy with a deterministic productive time between checkpoints. It is assumed that the program can be divided into equal parts, and that a checkpoint is placed at the end of each program part (except, maybe, the last program part.) For this model, the analysis of the program execution time has been considered in many papers (see, e.g.,

[Dud83, Gei88, Cof88, Gra92].) In [Gei88, Cof88], failures during a deterministic checkpoint duration were taken into account.

We use the same notation as in Section 7.3, with the added assumption that checkpoint duration is a random variable C, with a CDF $F_C(t)$, a pdf $f_C(t)$, and a LST $\phi_C(s) = E(e^{-sC})$. Unlike most previous models, we take into account the possibility of failures during checkpointing. A failure during a checkpoint causes a rollback to (i.e., a restart from) the previous checkpoint. Here we remark that regardless of how many times a program part (say, the i-th part) is repeated, the duration of its checkpoint remains fixed (say, c_i, where c_i is a realization of the random variable C.) We assume that failures occur according to a Poisson process at the same rate γ, also during checkpointing.

7.4.2 The Execution Time with Equidistant Checkpoints

In this section we derive the LST of the execution time of a program, when checkpoints are equally spaced with respect to the productive processing time. We also obtain an expression for its expectation and discuss the optimization with respect to the number of checkpoints in the program.

Let x be the total processing requirement of the program, which is divided into, say, n equal parts, and denote by $T(x, n)$ its total execution time. Note that there is a checkpoint at the end of each program part, except the last, thus a total of $n-1$ checkpoints. The following theorem gives an expression for the LST of $T(x, n)$.

THEOREM 2. With $n-1$ equidistant checkpoints, the LST of the program execution time in the presence of failures is given by

$$\begin{aligned}\phi_T(s,x,n) &= \left[\frac{(s+\gamma)\phi_C(s+\gamma)e^{-(s+\gamma)x/n}}{s+\gamma(1-\phi_R(s)(1-\phi_C(s+\gamma)e^{-(s+\gamma)x/n}))}\right]^{n-1} \\ &\quad \times \left[\frac{(s+\gamma)e^{-(s+\gamma)x/n}}{s+\gamma(1-\phi_R(s)(1-e^{-(s+\gamma)x/n}))}\right]. \end{aligned} \tag{7.3}$$

Proof: Each of the first $n-1$ program parts requires $x/n+C$ units of un-interrupted processing time to complete. The execution times of the first $n-1$ program parts are independent and identically distributed (i.i.d) random variables, each is given by $T(x/n+C)$. Note that the checkpoint duration C is a random variable; however, it is fixed for a given program part. Following similar steps as those used in the proof of Theorem 1, we obtain the following for the LST of $T(x/n+C)$

$$\phi_T(s, x/n+C) = \frac{(s+\gamma)\phi_C(s+\gamma)e^{-(s+\gamma)x/n}}{s+\gamma(1-\phi_R(s)(1-\phi_C(s+\gamma)e^{-(s+\gamma)x/n}))}.$$

The last program part requires x/n units of un-interrupted processing time (it does not include a checkpoint), and its execution time is given by $T(x/n)$. The LST of $T(x/n)$ is obtained directly from Equation 7.1

$$\phi_T(s, x/n) = \frac{(s+\gamma)e^{-(s+\gamma)x/n}}{s+\gamma(1-\phi_R(s)(1-e^{-(s+\gamma)x/n}))}.$$

The total program execution time is the sum of n independent random variables (corresponding to the execution times of the n parts.) The first $n-1$ of which are identically distributed and having the LST $\phi_T(s, x/n+C)$, and the last one having the LST $\phi_T(s, x/n)$. It follows that

$$\phi_T(s, x, n) = \left[\phi_T(s, x/n + C)\right]^{n-1} \left[\phi_T(s, x/n)\right],$$

and hence Equation 7.3. □

The LST in Equation 7.3 can be inverted by inspection or numerically to obtain the pdf $f_T(t, x, n)$. The first and higher moments of $T(x, n)$ can be obtained from its LST. Of particular interest is the expected program execution time, $E(T(x, n))$, which is given in the following corollary.

COROLLARY 2. With $n-1$ equally spaced checkpoints, the expected program execution time in the presence of failures is given by

$$E(T(x,n)) = (\frac{1}{\gamma} + E(R)) \left[(n-1)(\phi_C(-\gamma)e^{\gamma x/n} - 1) + (e^{\gamma x/n} - 1)\right]. \tag{7.4}$$

Proof: The proof follows directly from $E(T(x,n)) = \frac{-\partial \phi_T(s,x,n)}{\partial s}|_{s=0}$, or by following the same arguments as in Theorem 2. □

Assuming that $E(T(x,n))$ is a non-concave function of n (related convexity properties have been discussed in [Cof88]), then equidistant checkpointing is beneficial only if $E(T(x,2) < E(T(x,1)) = E(T(x))$, i.e., $e^{\gamma x/2}(\phi_C(-\gamma) + 1 - e^{\gamma x/2}) < 1$. Since C is a non-negative random variable, $\phi_C(-\gamma) \geq 1$, and the above inequality does not hold for $x \to 0$, i.e., checkpointing is not beneficial for a sufficiently small x.

If we fix the length of each program part to some constant, say, τ, then increasing the program processing requirement x (provided that $n = \frac{x}{\tau}$ remains an integer) would mean increasing the number of checkpoints in the program $(n-1)$. In this case, Equation 7.4 may be written as follows

$$E(T(x, x/\tau)) = (\frac{1}{\gamma} + E(R)) \left[(\frac{x}{\tau} - 1)(\phi_C(-\gamma)e^{\gamma\tau} - 1) + (e^{\gamma\tau} - 1)\right]. \tag{7.5}$$

Note that, for a fixed τ, $E(T(x, x/\tau))$ grows only linearly with the total program processing requirement x. (Recall that, without checkpointing, $E(T(x))$ grows exponentially with x, see Equation 7.2.)

Let $\hat{\tau}$ be the optimal length of a program part which minimizes the expected program execution time. It can be shown that $E(T(x, x/\tau))$ is a convex function of τ, and, hence, $\hat{\tau}$ can be determined by (numerically) solving the equation $\frac{d}{d\tau}(E(T(x, x/\tau))) = 0$.

For a sufficiently large x, we have the following approximation

$$E(T(x, x/\tau)) \approx (\frac{1}{\gamma} + E(R))(\phi_C(-\gamma)e^{\gamma\tau} - 1)\ \frac{x}{\tau}. \tag{7.6}$$

An approximation to $\hat{\tau}$ can be determined by solving the equation $\phi_C(-\gamma)e^{\gamma\hat{\tau}}(1-\gamma\hat{\tau}) = 1$. It is interesting to note that $\hat{\tau}$ is independent of x, however, it does depend on the failure rate γ and the distribution of checkpoint duration, through its LST $\phi_C(-\gamma)$. For a sufficiently small γ, further approximation yields the following simple expression for the optimal inter-checkpoint interval $\hat{\tau} \approx \frac{1}{\gamma}\sqrt{2(1-\frac{1}{\phi_C(-\gamma))}}$. Let $\hat{n} = \lfloor \frac{x}{\hat{\tau}} \rfloor$, then the optimal number of checkpoints is given by $max(0, \hat{n}-1)$.

7.5 CHECKPOINTING IN MODULAR PROGRAMS

It is not always possible to place checkpoints equally spaced in a program, as this depends to a large extent on the structure of the program. In practice, many checkpoint placement schemes are based on the modularity of computation in a program (see, e.g., [Cha72b, Lon92].) This is due to the fact that the end of modules often constitute natural points to perform acceptance tests and to place checkpoints at a relatively low cost. For such schemes, there are only some (candidate) points in the program where checkpoints may be placed. (In [Tou84], the problem of optimal placement of checkpoints at such points is considered.) For example, a program may be composed of several modules or blocks (each requiring a random processing time to complete) and checkpoints may be placed only at the end of each module, except the last one. In such cases, the productive time between checkpoints (corresponding to the processing requirement of one or more modules) is a random variable. If the distribution of this random variable is known (by means of measurements, approximations, or otherwise), then it is of much interest to use it in models for the analysis of the program execution time.

In this section we consider a new model for the analysis of the program execution time in the presence of failures, with generally distributed productive time between checkpoints. More specifically, the model we consider here is the same as that in Section 7.4, except that the productive time between checkpoints (τ) is assumed to be generally distributed (rather than deterministic) with a CDF $F_\tau(t)$, a pdf $f_\tau(t)$, and a LST $\phi_\tau(s)$. This corresponds to placing checkpoints in a program at the end of modules having generally distributed processing time requirement.

There are two measures of interest here. The first is the execution time of a program consisting of, say, n modules; we denote this by $T_{mod}(n)$. The second is the execution time to complete a given amount of productive processing, say, x; which, as before, is denoted by $T(x)$. The analysis in this section is carried out to derive the expectation of the these two measures; however, the same approach can also be used to derive their distributions.

7.5.1 The Execution Time of a Modular Program

In this section we consider a model for the execution time of a modular program with checkpoints. The analysis of this model is useful for the evaluation of schemes in which checkpoints are placed at the program level, and based on the modularity of its computation. In the following theorem we give the expected execution time of a program consisting of n modules, each requiring an independent and identically (generally) distributed productive processing time and ending with a checkpoint (no checkpoint is placed at the end of the last module.)

THEOREM 3. With modular checkpointing, the expected execution time of a program consisting of n modules in the presence of failures is given by

$$E(T_{mod}(n)) = (\frac{1}{\gamma} + E(R)) \left[(n-1)(\phi_C(-\gamma)\phi_\tau(-\gamma) - 1) + (\phi_\tau(-\gamma) - 1)\right]. \quad (7.7)$$

Proof: Follows directly from Equation 7.4 of Corollary 2, after replacing the productive time between checkpoints, x/n, by τ and unconditioning on τ. □

For a deterministic productive time between checkpoints such that $\tau = x/n$, $\phi_\tau(-\gamma) = e^{\gamma x/n}$ and Equation 7.7 reduces to Equation 7.4. The following corollary specializes the above result to the case when the productive time between checkpoints is exponentially distributed with a mean equal to α^{-1}. In this case $f_\tau(t) = \alpha e^{-\alpha t}$ and $\phi_\tau(s) = 1/(\alpha + s)$.

COROLLARY 3. For exponential productive time between checkpoints, Equation 7.7 reduces to

$$E(T_{mod}(n)) = (\frac{1}{\gamma} + E(R)) \left[(n-1)(\frac{\phi_C(-\gamma)}{\alpha - \gamma} - 1) + (\frac{1}{\alpha - \gamma} - 1)\right]. \quad (7.8)$$

Proof: Direct substitution of $\phi_\tau(-\gamma) = \frac{1}{\alpha - \gamma}$ in Equation 7.7. □

Note that, for checkpointing in a given modular program (as considered here), optimization with respect to the number of modules (n) is not straightforward. This is due to the fact that, in general, changing n also implies a change in the distribution of the processing requirement of each module $(F_\tau(t))$.

7.5.2 The Execution Time of a Given Processing Requirement

In this section we consider the execution time of a given amount of productive processing in a program, with a checkpoint placed at the end of each module. This model is a generalization of that in Section 7.4, since here we assume that the productive processing between checkpoints is generally distributed (rather than deterministic). The analysis of this model is useful for the evaluation of non-deterministic checkpointing schemes, in which the productive processing times between checkpoints are independent and identically distributed random variables. In the following theorem we give the expected execution time to complete x units of productive processing time.

THEOREM 4. With generally distributed productive time between checkpoints, the expected execution time to complete x units of productive processing time in the presence of failures is given by the solution of the following integral equation

$$E(T(x)) = (\frac{1}{\gamma} + E(R)) \left[\phi_C(-\gamma) \int_{h=0}^{x} e^{\gamma h} dF_\tau(h) + e^{\gamma x}(1 - F_\tau(x)) - 1 \right] + \int_{h=0}^{x} E(T(x-h)) dF_\tau(h). \quad (7.9)$$

Proof: Let H be the productive time between checkpoints, then conditioning on $H = h$, we have

$$E(T(x))|_{H=h} = \begin{cases} (\frac{1}{\gamma} + E(R))(e^{\gamma x} - 1), & \text{if } h \geq x \\ (\frac{1}{\gamma} + E(R))(\phi_C(-\gamma)e^{\gamma h} - 1) + E(T(x-h)), & \text{if } h < x. \end{cases}$$

If $h \geq x$, then there are no checkpoints during the execution of x units of productive time. From Equation 7.2, it follows that the expected execution time is given by

$$E(T(x)) = (\frac{1}{\gamma} + E(R))(e^{\gamma x} - 1).$$

If $h < x$, then a checkpoint will occur before completing x units of productive time. In this case, the expected execution time to complete h units of productive time and C units of checkpointing time is given by (note that, although C is a random variable, it is fixed for a given checkpoint)

$$E(T(h + C)) = (\frac{1}{\gamma} + E(R))(\phi_C(-\gamma)e^{\gamma h} - 1).$$

After the checkpoint, h units of productive time are completed and the expected remaining execution time is given by $E(T(x - h))$.

Unconditioning on H, we get $E(T(x)) = \int_{h=0}^{\infty} E(T(x))|_{H=h} \; dF_\tau(h)$, from which Equation 7.9 follows. □

For a deterministic productive time between checkpoints such that $\tau = x/n$, it can be easily shown that Equation 7.9 reduces to Equation 7.4. The following corollary specializes the above result to the case when the productive time between checkpoints is exponentially distributed with a mean equal to α^{-1}.

COROLLARY 4. For exponential productive time between checkpoints, an explicit solution for Equation 7.9 is given by

$$E(T(x)) = (\frac{1}{\gamma} + E(R)) \left(\frac{\gamma + \alpha(\phi_C(-\gamma) - 1)}{(\alpha - \gamma)^2} \right) \times \left(\alpha(\alpha - \gamma)x + \gamma(e^{-(\alpha-\gamma)x} - 1) \right). \quad (7.10)$$

Proof: Substituting for $dF_\tau(h) = \alpha e^{-\alpha h} dh$ and for $F_\tau(x) = (1 - e^{-\alpha x})$ in Equation 7.9, we get

$$\begin{aligned} E(T(x)) &= (\frac{1}{\gamma} + E(R)) \left(\frac{\gamma + \alpha(\phi_C(-\gamma) - 1)}{\alpha - \gamma} \right) (1 - e^{-(\alpha-\gamma)x}) \\ &\quad + \alpha \int_{h=0}^{x} E(T(x-h)) e^{-\alpha h} dh. \end{aligned}$$

Define the Laplace transform $\tilde{T}(w) = \int_{x=0}^{\infty} e^{-wx} E(T(x)) dx$. After changing the order of integration in the above equation and some manipulation, we get

$$\begin{aligned} \tilde{T}(w) &= (\frac{1}{\gamma} + E(R)) \left(\frac{\gamma + \alpha(\phi_C(-\gamma) - 1)}{(\alpha - \gamma)^2} \right) \\ &\quad \times \left(\frac{\alpha(\alpha - \gamma)}{w^2} + \gamma(\frac{1}{w + \alpha - \gamma} - \frac{1}{w}) \right). \end{aligned}$$

Inverting $\tilde{T}(w)$ with respect to w yields Equation 7.10. □

Note that, for $\alpha = 0$ (i.e., no checkpointing), $E(T(x))$ from Equation 7.10 reduces to $(\frac{1}{\gamma} + E(R))(e^{\gamma x} - 1)$, in agreement with Equation 7.2. For a sufficiently large x, it can be shown that $E(T(x))$ is a convex function of α and that $\frac{dE(T(x))}{d\alpha}|_{\alpha=0} < 0$, i.e., checkpointing is beneficial. It can also be shown that the optimal checkpointing rate is higher than the failure rate γ. For a large x, and in the particularly interesting range, namely, $\alpha > \gamma$, we obtain the following approximation for $E(T(x))$

$$E(T(x)) \approx (\frac{1}{\gamma} + E(R)) \left(\frac{\gamma + \alpha(\phi_C(-\gamma) - 1)}{(\alpha - \gamma)} \right) \alpha x. \tag{7.11}$$

In the above approximation, $E(T(x))$ is a linear of x. Also, an approximation to the optimal checkpointing rate $\hat{\alpha}$ is independent of x and is given by

$$\hat{\alpha} \approx \gamma \left(1 + \sqrt{\frac{\phi_C(-\gamma)}{(\phi_C(-\gamma) - 1)}} \right). \tag{7.12}$$

Substituting for $\hat{\alpha}$ from Equation 7.12 in Equation 7.11 yields an approximation for the minimum expected execution time when x is large

$$E(T(x)) \approx (\frac{1}{\gamma} + E(R)) \gamma x \left(1 + 2(\phi_C(-\gamma) - 1) + 2\sqrt{\phi_C(-\gamma)(\phi_C(-\gamma) - 1)} \right). \tag{7.13}$$

The above approximations provide good estimates for a practical range of parameter values.

7.6 RANDOM CHECKPOINTING

In Section 7.2, we have described some situations in which checkpointing at the program level may be considered to occur randomly. For example, using an external trigger (e.g., a real-time clock [Li90] or a change in the system's workload) to determine checkpoint locations can result in non-deterministic checkpoint placements (relative to the computational progress) for different executions of the same program. As a consequence of non-determinism (or randomness), checkpointing may also occur during recovery (reprocessing) after failures. Since failures are usually more likely to occur during recovery, this is an important feature that is worthwhile considering in a checkpointing strategy. This feature is taken into account in [Kul90] for modeling and analysis of random checkpointing.

7.6.1 The Model

In this section we describe a model of random checkpointing in a program that we consider for analysis. As in [Kul90], it is assumed that the system, on which the program is being executed, can be in one of three states; namely, *operational* (includes reprocessing), *checkpointing*, or *under-repair* (excludes reprocessing.) In this model, it is important to note that the two components of recovery time (i.e., repair and reprocessing) are separated in two states; namely, repair time in the under-repair state and reprocessing time which constitutes a part of the operational state. When the system is in the operational state, failures and checkpoints occur according to a Poisson process at rates γ and α, respectively. A failure takes the system to the under-repair state for a random repair time R (excludes reprocessing), after which the system returns to the operational state, and the program is restarted from the most recent checkpoint. A checkpoint takes the system to the checkpointing state for a random checkpointing time C, after which the system returns to the operational state, and the program execution continues. At a checkpoint, the status of the program is saved, so that all useful processing done thus far is never lost again.

In this section, we extend the model in [Kul90] to allow for the possibility of failures (at the same rate γ) during checkpointing. A failure in the checkpointing state takes the system to the under-repair state for a random repair time R, after which the system returns to the operational state, and the program is restarted from the previous (successful) checkpoint. Unlike the models considered in Sections 7.4 and 7.5, in the present model, it is appropriate to assume that subsequent checkpoints (including those next to a failed checkpoint) have different durations (i.e., different i.i.d. realizations of the random variable C.)

7.6.2 The Execution Time of a Given Processing Requirement

In this section we consider the execution time of a given amount of productive processing in a program with random checkpointing. We follow a similar approach as that in [Kul90] to derive the LST of the execution time and its expectation. We also discuss the minimization of the expected execution time with respect to the checkpointing rate α.

Let $\acute{C}$ be a random variable which denotes the holding time in the checkpointing state; it is the minimum of the two random variables, C and the time to next failure (the latter being exponentially distributed with a mean γ^{-1}.) It can be shown that the LST of $\acute{C}$ is given by

$$\phi_{\acute{C}}(s) = \frac{\gamma + s\phi_C(s+\gamma)}{s+\gamma}. \tag{7.14}$$

It follows that the expected holding time in the checkpointing state is given by $E(\acute{C}) = (1-\phi_C(\gamma))/\gamma$.

In the presence of random failures and checkpointing, let $T(x)$ be a random variable which denotes the execution time of a program that requires x units of processing time. In the following theorem we give a closed form expression for the LST $\phi_T(s,x)$. We give only a starting recursive relation for $T(x)$ leading to the final result. The complete proof follows similar steps as those in [Kul90].

THEOREM 5. With random checkpointing, the LST of the program execution time in the presence of failures is given by

$$\begin{aligned}\phi_T(s,x) \quad = \quad & \left\{ \left[\frac{\gamma + \alpha\phi_{\acute{C}}(s)(1-\phi_C(\gamma))}{s+\alpha+\gamma} \right] \phi_R(s)(1 - e^{(s+\alpha+\gamma)x}) \right. \\ & \left. + e^{(s+\alpha+\gamma)x} \right\}^{1 - \frac{\alpha\phi_{\acute{C}}(s)\phi_C(\gamma)}{s+\alpha+\gamma-\left[\gamma+\alpha\phi_{\acute{C}}(s)(1-\phi_C(\gamma))\right]\phi_R(s)}}.\end{aligned} \tag{7.15}$$

Proof: Let H be the holding time in the operational state until the first event (a failure or a checkpoint), then conditioning on $H = h$, we have

$$T(x)|_{H=h} = \begin{cases} x, & \text{if } h \geq x \\ h + \acute{C} + T(x-h), & \text{if } h < x \text{ and event is a successful checkpoint} \\ h + \acute{C} + R + T(x), & \text{if } h < x \text{ and event is a failed checkpoint} \\ h + R + T(x), & \text{if } h < x \text{ and event is a failure.} \end{cases}$$

If $h \geq x$, then the program will complete in x units of time. If $h < x$ and the event is a checkpoint, then there is a holding time $\acute{C}$ in the checkpointing state, followed by one of two possibilities. If the checkpoint is successful (with a probability $\phi_C(\gamma)$), then program execution continues with $x-h$ units of productive processing time remaining to complete. Otherwise, the checkpoint fails (with a probability $(1-\phi_C(\gamma))$), and the system spends R time units in repair before restarting the program, with x units of productive processing time remaining to complete. If $h < x$ and the event is failure, then the system spends R time units in repair before restarting the program, with x units of productive processing time remaining to complete.

The proof can be completed by unconditioning on H (using $f_H(h) = (\alpha+\gamma)e^{-(\alpha+\gamma)h}$) and following similar steps as those in [Kul90]. □

The LST in Equation 7.15 can be inverted numerically to obtain $f_T(t,x)$. The first and higher moments of $T(x)$ can be obtained from its LST. Of particular interest is the expected program execution time, $E(T(x))$, which is given in the following corollary.

COROLLARY 5. With random checkpointing, the expected program execution time in the presence of failures is given by

$$E(T(x)) = a((\alpha+\gamma)x + \ln b(x)), \tag{7.16}$$

with

$$a = \frac{1 + \alpha E(\acute{C}) + (\alpha(1-\phi_C(\gamma)) + \gamma)E(R)}{\alpha\phi_C(\gamma)},$$

and

$$b(x) = \frac{\alpha\phi_C(\gamma) + (\alpha(1-\phi_C(\gamma)) + \gamma)e^{-(\alpha+\gamma)x}}{\alpha+\gamma}.$$

Proof: Follows directly from the LST in Equation 7.15, using the relation $E(T(x)) = \frac{-\partial\phi_T(s,x)}{\partial s}|_{s=0}$. □

Again, checkpointing is beneficial only if $E(T(x))$ is less than the expected execution time without checkpointing, i.e., only if $E(T(x)) < (\frac{1}{\gamma} + E(R))(e^{\gamma x} - 1)$. This inequality may not hold for a sufficiently small x. For a sufficiently large x, $b(x)$ approaches $\frac{\alpha\phi_C(\gamma)}{(\alpha+\gamma)}$, which, for a small γ, approaches 1. In this case, $E(T(x))$ may be approximated by the following linear function of x,

$$E(T(x)) \approx a(\alpha+\gamma)x. \tag{7.17}$$

It follows that, for large x, an approximation to the optimal checkpointing rate $\hat{\alpha}$, which minimizes $E(T(x))$, is independent of x and is given by

$$\hat{\alpha} \approx \sqrt{\frac{\gamma(1+\gamma E(R))}{E(\acute{C}) + (1-\phi_C(\gamma))E(R)}}. \tag{7.18}$$

Substituting for $\hat{\alpha}$ from Equation 7.18 in Equation 7.17 yields an approximation for the minimum expected execution time for large x

$$E(T(x))|_{\alpha=\hat{\alpha}} \approx \frac{x}{\phi_C(\gamma)}\left(\sqrt{1+\gamma E(R))} + \sqrt{\gamma(E(\acute{C}) + (1-\phi_C(\gamma))E(R))}\right)^2. \tag{7.19}$$

The above approximations are quite useful as they could provide very good estimates for a practical range of parameter values.

7.7 CONCLUSIONS

Techniques for fault-tolerance are usually designed with their main goal is to satisfy certain dependability requirements of the system. This is often done at the expense of additional cost and/or degraded performance. It is therefore important to evaluate fault-tolerance procedures not only based on their dependability aspects [Lap84] but also based on their

cost/performance aspects (some dependability and performance evaluation techniques [Tri82] are discussed in chapters 5 and 6 of this book).

Checkpointing is a commonly used technique for fault-tolerance in a variety of software applications and information systems. It involves intermittently saving sufficient information, which is used to avoid a complete loss of useful processing (work) at the occurrence of a transient failure. Typically, the main goal of checkpointing is twofold: to satisfy the reliability requirements of the system in the presence of failures, and to do so with a minimum performance sacrifice. If checkpoints were without a cost, then more checkpoints would mean meeting the reliability requirements at a better performance. However, this is rarely the case, and there is typically a trade off, i.e., there is an optimal checkpointing strategy (frequency) which optimizes an appropriate performance measure. In information (database) systems, the objective is usually to maintain the integrity of information, while maximizing its long run availability. In software applications, the objective is usually to maintain the correctness of a long executing program, while minimizing its expected execution time or keeping it within acceptable bounds.

In this chapter, our main focus has been the modeling of checkpointing and rollback recovery in a program. We have considered some basic models for the analysis of the program execution time in the presence of failures, with some commonly used checkpointing strategies; namely, the (well-known) equidistant strategy, a strategy used in modular programs and what we call "random" strategy. The latter maybe used to model strategies such as "global" checkpointing [Kan78] or "dynamic" checkpoint insertion schemes [Li90]. In all models considered in this chapter, we have assumed that failures may occur not only during recovery, but also during checkpointing (resulting in a rollback to the previous successful checkpoint), thus making an important extension to some existing models. We have also introduced and analyzed a new model for checkpointing in modular programs, where checkpoints may be placed only at the boundaries between program modules. The processing time requirement of a program module is assumed to be generally distributed. Although we have assumed a homogeneous Poisson failure process, most of the analysis can be carried out for other failure distributions; however, the final results may no longer turn out to be explicit or appear in a compact form. (A model with age dependent failures is considered in [Cof88].) The results of our analysis include closed form expressions for the LST (Laplace Stieltjes transform) of the program execution time and/or its expectation. The first and higher moments can be directly determined from the derivatives of the LST; however, numerical inversion of the LST is usually necessary to obtain the complete distribution of the program execution time. A conclusion which generally holds is that, with checkpointing, the expected execution time of a program grows only linearly (instead of exponential growth without checkpointing) with its processing requirement. It is noted that checkpointing may not always be beneficial, particularly for programs with sufficiently small processing requirement. Otherwise, there is usually an optimal (with respect to some criterion) checkpointing frequency, which can be determined either analytically or numerically. For programs with sufficiently large processing requirement, the optimal checkpointing frequency, which minimizes the expected execution time, is independent of the program processing requirement and is equal to the optimal checkpointing frequency which maximizes the long run system availability.

Needless to say, with much already accomplished, still there are many useful extensions and practical problems that need to be addressed. A dynamic programming approach to the modeling and analysis of checkpointing and recovery has been considered at the program level [Tou84] and at the system level [Lec88], and it typically yields an algorithmic solution.

This approach is very promising and should be explored further, as it is capable of handling more realistic models with less restrictive assumptions.

In most previous work, as well as here, it has been assumed that failures are detected as soon as they occur. In reality, a failure may go undetected, leading to incorrect execution of the program. Or, as often is the case, there is a latency between failure occurrence and detection, which may cause the program to rollback to earlier checkpoints. Some models have considered these effects, for example, when combining checkpointing with instruction retries [Kor86], also in real-time [Shi87] and concurrent processing [Tha86] environments. It is of much interest to better understand the impact of such a latency in various systems and under different scenarios.

Also when tolerated, failures are likely to cause some sort of system degradation (for example, the unavailability of some processors in a multi-processor system). This may affect the processing (or reprocessing) speed after failures and until further repair action is taken to bring the system back to its full processing capacity. In some systems, another possible scenario is that checkpointing may be performed in the background, thus allowing productive processing (perhaps at a lower speed) to continue during checkpointing. It is certainly important to consider checkpointing models that take into account such a degradable/repairable or concurrent behavior, as this will largely influence the optimal checkpointing strategy. Models such as those considered in [Kul90] and in Section 7.6 of this chapter may be extended by adding more states with appropriate rewards (corresponding to processing speeds) to account for such behaviors.

Many other complications and problems arise if the program is running in a concurrent or a distributed environment (see [Cha85, Jon90, Lon91, Str85] and many others.) On the other hand, parallel (or multi-processor) computing systems provide hardware redundancy which can be exploited to achieve fault-tolerance with less performance degradation. In these systems, the same program is run on more than one processor. Checkpoints are used to detect failures (by comparing the states of the processors) and to reduce recovery time. Roll-forward recovery schemes have been suggested to take advantage of this hardware redundancy (see, for example, [Lon90, Pra92].) Similarly, software redundancy schemes can be used to achieve fault-tolerance by masking software (program) failures (see [Avi85, Tso87].) So far, only little has been done to model program checkpointing and recovery schemes and to understand their effects in such contexts (see, e.g., [Koo87, Nic90a, Shi84, Ziv93].)

Finally, there is growing interest and need for developing robust and effective checkpointing/recovery schemes at different levels of software applications running on various architectures and platforms. It is also apparent that these schemes are becoming increasingly complex and sophisticated, thus requiring more research and analysis to help abstract the main issues and understand the various trade-offs that are typically involved in their design.

REFERENCES

[And81] T. Anderson and P. Lee. *Fault Tolerance—Principles and Practice*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Avi85] A. Avižienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, 1985.

[Bac81a] F. Baccelli. Analysis of a service facility with periodic checkpointing. *Acta Informatica*, 15(1):67–81, 1981.

[Bac81b] F. Baccelli and T. Znati. Queueing algorithms with breakdowns in database modeling. In F.J.