Analytic Models of Checkpointing for Concurrent Component-Based Software Systems

Noor Bajunaid
Computer Science Department
George Mason University
Fairfax, VA 22030
nbajunai@masonlive.gmu.edu

Daniel A. Menascé
Computer Science Department
George Mason University
Fairfax, VA 22030
menasce@gmu.edu

ABSTRACT

Checkpointing and rollback is a key mechanism used to improve the reliability of software systems. The benefits of this mechanism can be offset by the overhead of checkpointing when the failure rate is low. The problem of developing analytic models of rollback and checkpointing has been continuously addressed for over four decades using different assumptions. This paper examines the problem under a more realistic angle, i.e., one in which there are several software components sharing resources (e.g., processors and I/O devices) among themselves and with the checkpointing processes. Additionally, the paper allows for different components to have different computing, rollback, and checkpointing demands, as well as different failure distributions. Our models also allow for various checkpointing processes to be executing concurrently to checkpoint the state of different software components. The analytic models developed here combine Markov Chains and Queuing Networks and allow us to compute the following metrics: (1) average time needed by a component to complete its execution, (2) average throughput of a component, (3) availability of a component, and (4) checkpointing overhead. The models were validated through extensive simulation and experimentation.

CCS Concepts

•Software and its engineering → Software performance; Software reliability; Checkpoint / restart;

Keywords

Checkpointing; concurrent components; analytic models; Markov chains; queuing networks

1. INTRODUCTION

One of the key mechanisms used to improve the reliability of software systems is checkpointing and rollback, which allows for a software component to store the state of its computation at regular intervals to reduce the amount of work to be redone in the case of a failure [27]. Without checkpointing, a software component (referred to as component hereafter) has to restart the computation from the beginning, which could be a problem for long computing tasks. With checkpointing, the computation resumes from the latest checkpoint after the computation is restored to that state.

The benefits of checkpointing can be offset by the overhead of checkpointing when the failure rate is low. The problem of developing analytic models for rollback and checkpointing has been continuously addressed for over four decades using different assumptions. This paper examines the problem under a more realistic angle, i.e., one in which there are several software components sharing resources among themselves and with the checkpointing processes. Additionally, the paper allows for different components to have different computing, rollback, and checkpointing demands as well as different failure rates. Our models also allow for various checkpointing processes to be executing concurrently to checkpoint the state of different software components. The analytic models developed here combine Markov Chains and Queuing Networks (QN) and allow us to compute the following metrics: (1) average time needed by a component to complete its execution, (2) average throughput of a component, (3) availability of a component, and (4) checkpointing overhead. The models were validated through simulation and experimentation.

As discussed in Section 4, there is a vast body of work that presents analytic models for obtaining the checkpointing interval that optimizes a variety of metrics. Some examples include: minimize total execution time, maximize availability, maximize a job's progress, and minimize the overhead generated by checkpointing and wasted work due to rollback. A comprehensive and relatively recent book by Katinka Wolter [25] contains a thorough description of many stochastic models for checkpointing, restart, and rejuvenation, including many discussed in the related work section of this paper and other novel models introduced by Wolter. However, the aforementioned models do not consider contention among components while executing or checkpointing.

Table 1 illustrates how the optimal duration of the interval between checkpoints, T_{opt}, varies for each of the metrics mentioned above and for different numbers of concurrent components ($n = 1, 4, 8$, and 16). The results in the table were obtained with the models described in this paper and assume Poisson failures at a rate of 0.005 failures per time unit. The models presented in this paper allow for

ICPE '17 April 22–26, 2017, L'Aquila, Italy

DOI: 10.1145/3030207.3030209

Table 1: Optimal checkpointing interval (T_{opt}) for different number of components (n) and $\lambda = 0.005$.

Metric	$n = 1$	$n = 4$	$n = 8$	$n = 16$
Execution time	10	7	7	7
Availability	10	8	7.5	7
Relative progress	10	7	7	7
Overhead	10	15	20	20

any failure distribution. The results in the table assume that the checkpointing time (without contention) is equal to 0.25 time units. According to a well-known result by Young [26], which does not consider contention, the value of T_{opt} that minimizes execution time is $\sqrt{2 \times 0.25/0.005} = 10$ time units. This value coincides with the value in Table 1 for $n = 1$. But, for other values of n, the value of T_{opt} is no longer the same as the one when contention is not considered. The table clearly shows that T_{opt} varies with the number of components running concurrently for the other metrics.

The main contribution of this paper is the development of analytic models based on a combination of Markov Chains and QNs to obtain the response time, throughput, and availability of *concurrent* and *heterogeneous* component-based software systems that use checkpointing. Differently from prior work, our models take into account contention for processors and storage devices by concurrent processes during their computation and checkpointing phases. Our models also allows for different software components to have different compute and I/O characteristics as well as different failure distributions.

This paper is organized as follows. Section 1 discusses some core results common to the two models described in the paper and then presents two checkpointing models: (1) a model that considers *homogeneous* components (all components have similar processing, I/O, and failure characteristics) and (2) a model for the case of *heterogeneous* components (different components may have different processing, I/O, and failure characteristics). Section 2 presents several numerical results using the models of the previous section. Section 3 discusses a validation of the models. The next section discusses related work and Section 5 presents some concluding remarks and discusses future work. Modeling Checkpointing Consider that a computing node runs n software components, $C_1, \cdots$
$, C_k, \cdots, C_n$, and that they request that their state be checkpointed every time they achieve a certain amount of computation. Thus, each component alternates between two modes: *computing* and *checkpointing*. In the former mode, a component is making forward progress towards its computational goal. In the latter, a checkpoint is being made of the component. The smaller the amount of computation before checkpoints, the faster is the recovery to the latest checkpoint at the expense of a higher checkpointing overhead. Conversely, a larger computing time before checkpointing decreases the checkpointing overhead but increases the time to return the computation to the failure point. The optimal checkpointing frequency depends on the: (1) failure rate, (2) computing time before checkpointing, and (3) time/overhead of taking checkpoints.

Checkpoint generation uses the same resources (i.e., processors and I/O devices) used by the components running at a computing node. We also assume that a component suspends its computation while checkpointing. So, each component alternates between computing and checkpointing. When a failure occurs while a component is computing, its state has to be restored to its latest checkpoint (i.e., a failure recovery) and the computation has to be restarted from that checkpointed state. In the process, all the computation since the latest checkpoint is wasted and has to be re-done.

Our models use the following assumptions:

- A1: Components can fail individually without affecting other components running on the same machine. Such failures may occur, for example, due to lack of memory, software bugs, or inability to allocate a needed resource.
- A2: One or more components may execute concurrently on the same machine.
- A3: Different components may have different processing and I/O demands. These demands are not deterministic but are expressed by their average values for each component.
- A4: Checkpoints for different components are not synchronized with those of other components and may be generated at different frequencies.
- A5: Different components may have different processing and I/O demands for generating checkpoints. These demands are not deterministic but are expressed by their average values for each component.
- A6: Different components may have different failure distributions.
- A7: Components and their checkpointing processes share resources (e.g., processors and I/O devices) on the machine they run and therefore there is *contention* for these resources by these processes.
- A8: Similarly to the vast body of literature on checkpointing modeling (see e.g. [17]) we assume:
  - failures are immediately detected,
  - no failures occur during checkpointing or during rollback, a reasonable assumption since the time to checkpoint and rollback is typically smaller than the MTTF.

Table 2 summarizes the notation used in this paper. Before we discuss in subsections 1.2 and 1.3 the *homogeneous component* and *heterogeneous component* models, we present results that are common to both.

1.1 Core Results

Figure 1 shows what can happen with a component between two checkpoints assuming two failures occur before the component is able to complete T time units of computation. In the figure, the first failure occurs after the component has processed for $W_1 < T$ time units. Then, the component spends a time equal to RT to rollback to the latest checkpoint. RT includes CPU and I/O time. The component then restarts its computation and after computing for $W_2 < T$ time units, a second failure occurs. Another period

Table 2: Notation used in the checkpointing models

Notation	Meaning
n	number of components
$C_{\text{CPU},k}$	avg. CPU time to checkpoint component k
$C_{\text{I/O},k}$	avg. I/O time to checkpoint component k
$E_{\text{CPU},k}$	avg. CPU time needed to execute component k
$E_{\text{I/O},k}$	avg. I/O time needed to execute component k
E_k	avg. time (CPU and I/O) to execute component k. $E_k = E_{\text{CPU},k} + E_{\text{I/O},k}$
$T_{\text{CPU,k}}$	CPU time between consecutive checkpoints of component k
$T_{\text{I/O,k}}$	I/O time between consecutive checkpoints of component k
T_k	total time between consecutive checkpoints of component k. This time, equal to $T_{\text{CPU,k}} + T_{\text{I/O,k}}$, determines when the next checkpoint has to occur.
$\hat{T}_k$	time to complete E_k after the last checkpoint of component k.
$\tilde{F}_k$	r.v. that represents the failure instant of component k after its last rollback or latest checkpoint when it is in computing mode
$f_{\tilde{F}_k}(x)$	pdf of $\tilde{F}_k$
q_k	probability that a failure occurs while component k is computing.
$\tilde{W}_{\text{CPU},k}$	r.v. that denotes the amount of computation to be re-done (i.e., wasted) per failure of component k
$\tilde{W}_{\text{I/O},k}$	r.v. that denotes the I/O time to be re-done (i.e., wasted) per failure of component k
$\tilde{W}_k$	r.v. that denotes the CPU and I/O time to be re-done (i.e., wasted) per failure of component k. $\tilde{W}_k = \tilde{W}_{\text{CPU},k} + \tilde{W}_{\text{I/O},k}$.
$\text{RT}_{\text{CPU},k}$	average CPU time needed to rollback component k to the latest checkpoint after a failure
$\text{RT}_{\text{I/O},k}$	average I/O time needed to rollback component k to the latest checkpoint after a failure

equal to RT ensues. The component then restarts its computation and this time it succeeds in completing the T time units of computation before the next checkpoint. It should be noted though that the time intervals C, W1, W2, RT and T in Fig. 1 represent time intervals for one component when there is no contention with other components (i.e., the case of $n = 1$). In the general case of $n \geq 1$ treated in this paper, these time intervals are elongated by the contention effect.

The number of checkpoints NX_k expected to be performed by component k is $\lfloor E_k/T_k \rfloor$, and the remaining amount of work $\hat{T}_k$ to be performed by component k after the last checkpoint is $\hat{T}_k = E - (T.NX_k)$.

The probability of exactly j failures experienced by component k between two checkpoints is $q_k^j(1-q_k)$ assuming independence among failures, where $q_k = \int_0^{T_k^-} f_{\tilde{F}_k}(x)\ dx$. If the failure model for component k is Poisson, i.e., $\tilde{F}_k$ is exponentially distributed with parameter λ_k, then $q_k = 1 - e^{-\lambda_k\ T_k}$. Thus, the average number of failures NF_k between consecutive checkpoints is

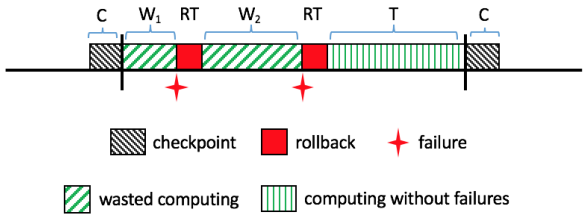

Figure 1: Timeline between two consecutive checkpoints.

$$NF_k = \sum_{j=0}^{\infty} j \times q_k^j(1-q_k) = \frac{q_k}{1-q_k}. \tag{1}$$

The average effective CPU time, $T^{\text{eff}}_{\text{CPU},k}$, that a component k spends computing and rolling back from failures between checkpoints is the product of the average number of failures NF_k by the sum of the average wasted CPU time, $E[\tilde{W}_{\text{CPU}}]$, and the average CPU time, $\text{RT}_{\text{CPU},k}$, needed to recover from a failure, plus the time $T_{\text{CPU,k}}$ to execute a successful computation between checkpoints.

$$T^{\text{eff}}_{\text{CPU},k} = \frac{q_k[E[\tilde{W}_{\text{CPU},k}] + \text{RT}_{\text{CPU},k}]}{1-q_k} + T_{\text{CPU},k} \quad k = 1, \cdots, n. \tag{2}$$

Similarly, the effective I/O time, $T^{\text{eff}}_{\text{I/O},k}$, needed by component k to recover from failures and execute a successful computation between checkpoints is given by

$$T^{\text{eff}}_{\text{I/O},k} = \frac{q_k[E[\tilde{W}_{\text{I/O},k}] + \text{RT}_{\text{I/O},k}]}{1-q_k} + T_{\text{I/O},k} \quad k = 1, \cdots, n. \tag{3}$$

Note that $T^{\text{eff}}_{\text{CPU},k}$ and $T^{\text{eff}}_{\text{I/O},k}$ do not include contention, which is accounted for by the models discussed in later subsections.

The average value of $\tilde{W}_k$, $E[\tilde{W}_k]$, is the average value of $\tilde{F}_k$ given that $\tilde{F}_k < T_k$ because when a failure occurs before the component completes its T_k seconds of computation, the computation up to the failure instant is wasted. The expression for $E[\tilde{W}_k]$ follows from the definition of conditional probability:

$$\begin{aligned} E[\tilde{W}_k] &= E[\tilde{F}_k \mid \tilde{F}_k < T_k] \\ &= \frac{\int_{x=0}^{T_k^-} x.f_{\tilde{F}_k}(x)dx}{\int_{x=0}^{T_k^-} f_{\tilde{F}_k}(x)dx}. \end{aligned} \tag{4}$$

For example, if $\tilde{F}_k$ is exponentially distributed (i.e., the failure model is Poisson), $f_{\tilde{F}_k}(x) = \lambda_k e^{-\lambda_k x}$ where λ_k is the failure rate. For this distribution, $E[\tilde{W}_k]$ can be computed as

$$E[\tilde{W}_k] = \frac{\int_{x=0}^{T_k^-} x.\lambda_k e^{-\lambda_k x}dx}{\int_{x=0}^{T_k^-} \lambda_k e^{-\lambda_k x}dx} \tag{5}$$

$$= \frac{\frac{1}{\lambda_k} - \left(\frac{1}{\lambda_k} + T_k\right) e^{-\lambda_k T_k}}{1 - e^{-\lambda_k T_k}}. \tag{6}$$

Note that the value of $E[\tilde{W}_k] \in (0, T_k)$. A value of 0 occurs when the failure occurs right at the beginning of the

computing segment and there is no need to re-do any computation after the component is restored to its most recent checkpoint. The value T_k occurs when the failure takes place at the very end of the computing segment. In this case, the entire computing segment has to be re-done after the component is restored to its latest checkpoint.

To derive expressions for $E[\tilde{W}_{\mathrm{CPU},k}]$ and $E[\tilde{W}_{\mathrm{I/O},k}]$, we note that the fraction of $E[\tilde{W}_k]$ wasted by the CPU is $E_{\mathrm{CPU},k}/E_k$. The same goes for wasted I/O time, as shown in Eqs. (7) and (8).

$$E[\tilde{W}_{\mathrm{CPU},k}] = \frac{E_{\mathrm{CPU},k}\ E[\tilde{W}_k]}{E_k}. \tag{7}$$

$$E[\tilde{W}_{\mathrm{I/O},k}] = \frac{E_{\mathrm{I/O},k}\ E[\tilde{W}_k]}{E_k}. \tag{8}$$

A variety of metrics can be computed using the analytic models described in the following two subsections. Of these, the following two are key because they capture the contention among the different components while computing, checkpointing, during rollback and recovery time.

- $\bar{r}_{c,k}$: average time it takes component k to complete its computation between consecutive checkpoints.
- $\bar{r}_{x,k}$: average time it takes component k to complete its checkpoint.

In order to execute T_k units of computing between checkpoints, a component needs $\bar{r}_{c,k}$ time units due to contention with other components performing computing and checkpointing, recovering from failures, and performing wasted work. And it will need $\bar{r}_{x,k}$ time units on average to perform each checkpoint. Thus, the average total time, $\bar{R}_{c,k}$, it takes a component to complete its computation taking into account contention with other components and checkpointing is

$$\bar{R}_{c,k} = [NX_k.(\bar{r}_{c,k} + \bar{r}_{x,k})] + (\hat{T}_k.\frac{\bar{r}_{c,k}}{T_k}). \tag{9}$$

where $\bar{r}_{c,k}/T_k$ is the inflation factor that must be applied to T_k due to failures and contention. We can also define the checkpointing overhead, Ov_k, of component k as the ratio between the time spent in checkpointing (including contention) and the total time (computing plus checkpointing).

$$Ov_k = \frac{NX_k.\bar{r}_{x,k}}{\bar{R}_{c,k}}. \tag{10}$$

The availability, A_k, of component k is defined as the fraction of time the component is doing useful work and is computed as the ratio between the time a component is either computing or waiting for resources divided by the total execution time. The term in square brackets in Eq. (11) represents the inflation factor that has to be applied to T_k^{eff} due to contention. This term is the ratio between the average time, $\bar{r}_{c,k}$, component k takes executing between checkpoints and the time the component would take with no contention. Note that inflation due to failures is not considered here. This definition of availability can be interpreted as the notion of achieved availability [8] where checkpointing corresponds to preventive maintenance and recovery time plus re-doing computation corresponds to corrective maintenance.

$$A_k = \frac{NX_k.T_k.[\bar{r}_{c,k}/(T^{\mathrm{eff}}_{\mathrm{CPU},k} + T^{\mathrm{eff}}_{\mathrm{I/O},k})]}{\bar{R}_{c,k}}. \tag{11}$$

We also define a metric called *relative progress* (RP_k) that indicates the relative progress of the computation, i.e., the relative amount of useful computation done by component k during a cycle:

$$RP_k = \frac{T_k}{\bar{r}_{c,k} + \bar{r}_{x,k}}. \tag{12}$$

Equations (9)-(12) require $\bar{r}_{c,k}$ and $\bar{r}_{x,k}$, which will be derived first for the case of homogeneous components and next for the case of heterogeneous components in the next two subsections, respectively.

1.2 Model for Homogeneous Components

The heterogeneous model can be used to solve systems with homogeneous components. However, the homogeneous model has a lower computational complexity. Under the homogeneous component assumption, we can drop the subscript k from the notation introduced in Table 2. All components have the same CPU and I/O demands for computing and checkpointing and the same failure distributions.

Because at any given time there may be a number of components in the computing mode and a number of components in the checkpointing mode, we model the system of components using a Markov Chain with states $0, 1, \cdots, v, \cdots, n-1, n$ where the state v indicates that there are v components performing their computation and $n - v$ components generating their checkpoints. The transition rate, μ_v, between states v and $v - 1$ is the rate at which one of the v components suspends its computation to start its checkpointing process. The transition rate, λ_v, between states v and $v + 1$ is the checkpointing completion rate for one of the components being checkpointed. The transition rates μ_v and λ_v are computed using a multiclass Queueing Network (QN) model as explained below. But first, we compute the probability p_v of being at state v of the Markov Chain. This probability is a function of λ_v, μ_v, and n.

An expression for p_v ($v = 0, \cdots, n$) is obtained by using the general birth-death equation for Markov Chains [15, 20]:

$$p_v = p_0 \prod_{i=0}^{v-1} \frac{\lambda_i}{\mu_{i+1}} \quad v = 1, \cdots, n \tag{13}$$

$$p_0 = \left[1 + \sum_{v=1}^{n} \Pi_{i=0}^{v-1} \frac{\lambda_i}{\mu_{i+1}}\right]^{-1}. \tag{14}$$

The values of p_v can be easily computed because the summation needed to compute p_0 is finite.

We now show how to compute the values of λ_v and μ_v. State v means that v components are computing and $n - v$ are checkpointing. We model this as a 2-class closed QN where the classes are: COMP and CHECK where COMP corresponds to segments of the computation between checkpoints and CHECK corresponds to the checkpointing processes. The population for the COMP class is v and for the CHECK class is $(n - v)$. Considering without loss of generality that there are two devices, CPU and I/O, the CPU and I/O service demands for the COMP class are $T^{\mathrm{eff}}_{\mathrm{CPU}}$ and $T^{\mathrm{eff}}_{\mathrm{I/O}}$, respectively, and for the CHECK class are C_{CPU} and C_{IO}, respectively.

The solution to the two-class closed QN described above can be obtained using the well-known Mean Value Analysis technique [20] and yields the following metrics:

- $X_{\mathrm{COMP}}(v)$: average throughput of component compu-

tation segments between checkpoints when there are v components in their computing phase.

- $X_{\text{CHECK}}(n-v)$: average throughput of component checkpoints, i.e., average rate at which checkpoints complete when there are $n-v$ concurrent checkpoints in progress.

Therefore, we can write the following two relationships:

$$\lambda_v = X_{\text{CHECK}}(n-v) \quad (15)$$

$$\mu_v = X_{\text{COMP}}(v). \quad (16)$$

We can also compute the average throughput, $\bar{X}_{\text{COMP}}$, of components completing their computation before starting checkpointing, and $\bar{X}_{\text{CHECK}}$, the average checkpointing throughput, as follows:

$$\bar{X}_{\text{COMP}} = \sum_{v=1}^{n} X_{\text{COMP}}(v)\ p_v \quad (17)$$

$$\bar{X}_{\text{CHECK}} = \sum_{v=0}^{n-1} X_{\text{CHECK}}(n-v)\ p_v. \quad (18)$$

The average number of components in the computing stage, $\bar{n}_c$, and the average number of components being checkpointed, $\bar{n}_x$, are:

$$\bar{n}_c = \sum_{v=1}^{n} v\ p_v \quad (19)$$

$$\bar{n}_x = n - \bar{n}_c. \quad (20)$$

The average time, $\bar{r}_c$, spent by a component between checkpoints and the average time, $\bar{r}_x$, a component takes to generate its checkpoint is computed with the use of Little's Law [18].

$$\bar{r}_c = \frac{\bar{n}_c}{\bar{X}_{\text{COMP}}} \quad (21)$$

$$\bar{r}_x = \frac{\bar{n}_x}{\bar{X}_{\text{CHECK}}}. \quad (22)$$

Thus, the values of $\bar{r}_c$ and $\bar{r}_x$ can be used to compute the average total time, $\bar{R}_c$, it takes a component to complete its computation taking into account contention with other components and checkpointing as indicated by Equation (9). Similarly, the checkpointing overhead, Ov, the avalilability A, and the relative progress RP can now be computed using Equations (10)-(12), respectively.

1.3 Model for Heterogeneous Components

This section generalizes the model presented in the previous section for the case in which the n components may have totally different characteristics. At any point in time, a component is either computing or checkpointing. So, the state of the system can be represented as $(e_1, \cdots, e_k, \cdots, e_n)$ where $e_k = 1$ if component k is computing and 0 if it is checkpointing. For example, if n is 3, the set of states is $\{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0),$ and $(1,1,1)\}$. Clearly, there are 2^n states. For convenience, we let $N = 2^n$. We number the states from 0 to $N-1$ and the state representation is the n-bit binary number corresponding to the decimal number of the state. We denote by $b(i)$ the binary representation of state i. For example, when $n = 3$, the states would be numbered from 0 to 7 and $b(6)$ is 110. We always number the bits in $b(s)$ from left to right starting from 1. Let $\mathcal{S}$ be the set of all states.

If we assume that the events of starting a checkpoint for one component and ending a checkpoint for the same or a different component cannot occur at exactly the same time, the only possible transitions between states are the ones that change only one of the e_k's of a state from 0 to 1 or from 1 to zero. We use the notation $\odot_k(b(s))$ to represent a function that flips the k-th bit in $b(s)$ leaving and all other bits unchanged. So, $\odot_4(0011011) = 0010011$.

We can now build a Markov Chain in which the states are the N states described above. Each state in $\mathcal{S}$ has n incoming transitions and n outgoing transitions to other states because outgoing and incoming transitions are associated to "flipping" one of the n bits of the state at a time. Figure 2 shows an example of this Markov Chain for the case of $n = 2$. The state transition rates will be explained later. States are identified by their decimal numbers $(0, \cdots, 3)$ and by their binary number. Note that for clarity, we have added after the 2-bit binary number another two bits to indicate which components are checkpointing in each state. These extra two bits are not needed to represent the state because this information can be derived by flipping all the bits of the state's binary representation.

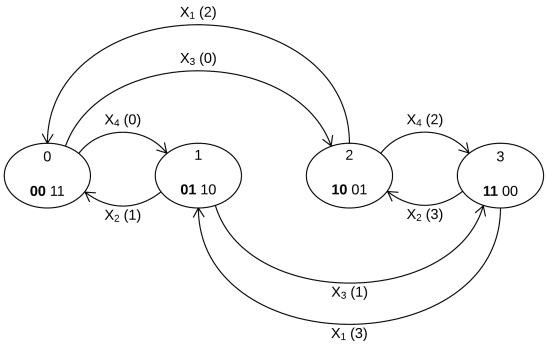

Figure 2: Example of Markov Chain for the heterogeneous case and $n = 2$.

In order to generate and solve the Markov Chain we need to write a set of balance equations and solve the resulting system of linear equations in which the variables are the probabilities $p_i, i = 0, \cdots, N-1$ of being in each state i. To find the state probabilities we need N equations, which can be obtained by applying the principle of *flow-in = flow-out* [15] to each state of the Markov Chain. This yields the following set of equations

$$\sum_{v \in \mathcal{S}_{\text{in}}(s)} p_v \times \mu_{v,s} - p_s \sum_{v \in \mathcal{S}_{out}(s)} \mu_{s,v} = 0 \quad \forall\ s \in \mathcal{S} \quad (23)$$

where $\mathcal{S}_{\text{in}}(s)$ is the set of all states from which there are transitions into state s, $\mu_{v,s}$ is the transition rate from state v into state s, $\mathcal{S}_{out}(s)$ is the set of states into which there are transitions from state s, and $\mu_{s,v}$ is transition rate from state s into state v. We can rewrite Eq. (23) using the state numbers from 0 to $N-1$. Thus, for each state $i, (i = 0, \ldots, N-1)$ we have the following equation:

$$\sum_{j=0}^{N-1} p_j \times \mu_{j,i} - p_i \sum_{j=0}^{N-1} \mu_{i,j} = 0. \quad (24)$$

The above equations are not linearly independent so we need to replace any of them with the equation that says that the sum of all probabilities is equal to 1 [15]:

$$\sum_{i=0}^{N-1} p_i = 1. \tag{25}$$

The matrix of transition probabilities, $\mathbf{P}$, is of the form

$$\mathbf{P} = \begin{pmatrix} \mu_{0,0} & \mu_{0,1} & \cdots & \mu_{0,N-1} \\ \mu_{1,0} & \mu_{1,1} & \cdots & \mu_{1,N-1} \\ \cdots & \cdots & \cdots & \cdots \\ \mu_{N-1,0} & \mu_{N-1,1} & \cdots & \mu_{N-1,N-1} \end{pmatrix} \tag{26}$$

The Markov Chain has the following properties:

- Property 1: $\mathcal{S}_{\text{in}}(s) = \mathcal{S}_{\text{out}}(s)\ \forall s \in \mathcal{S}$. This is true because, as indicated above, transitions between states can only occur when one and only one element e_k of the state changes from one to zero or from zero to one. So, $\mathcal{S}_{\text{in}}(s) = \mathcal{S}_{\text{out}}(s) = \{\odot_1(s)\} \bigcup \{\odot_2(s)\} \bigcup \cdots \{\odot_n(s)\}$.
- Property 2: $\mid \mathcal{S}_{\text{in}}\ (s) \mid = \mid \mathcal{S}_{\text{out}}\ (s) \mid = n\ \forall s \in \mathcal{S}$. This follows immediately from property 1 above.
- Property 3: $\mu_{i,i} = 0$ for $i = 0, \cdots, N-1$.
- Property 4: the bit that changed when going from state i to state j can be determined by performing a bitwise exclusive or operation between the binary representations of i and j. The result will have a bit equal to 1 where the change occurred and a zero in all other positions. This is true because there is only one bit change between i and j and that change is either from 0 to 1 or 1 to 0. All other bits remain the same; therefore the exclusive or between them is 0. Thus, the changed bit when going from state i to state j is the only bit equal to 1 in $b(i) \oplus b(j)$.
- Property 5: $\mu_{i,j} = 0$ if $b(i)$ and $b(j)$ differ by more than one bit because transitions can only occur at the beginning or end of the checkpointing of a single component.
- Property 6: Except for Eq. (25), most of the coefficients of the unknowns on equations (24) are zero. Because of properties 3 and 4 above, only n out of 2^n transitions rates are non-zero. Thus, the matrix of transition rates (see Eq. (26)) is largely sparse, a fact we benefit from when solving the Markov Chain.

To compute the transition rates $\mu_{i,j}$ we use a multiclass QN model similarly as before with the following characteristics: (1) There are $2n$ classes. Classes 1 through n represent the n components and the population of each of these classes is either 1 or 0, depending on the status of the component (i.e., computing (1) or checkpointing (0)). Classes $n+1$ through $2n$ correspond to the checkpointing process for each of the software components. The population of each of these classes is also 1 or 0 depending whether a component is checkpointing (1) or not (0). So, the total population of this QN model is equal to the number of components n. The population of class k plus the population of class $n+k$ is equal to 1 for $k = 1, \cdots, n$ because a component cannot be computing and checkpointing at the same time. Additionally, the service demands for the component classes may be different. A state $s \in \mathcal{S}$ represents the population of classes 1 through n of the QN. The population of the remaining classes is automatically derived from the population of the first n classes. For example, the state (1, 1, 0, 1, 0) for a 5-component system indicates that components 1, 2, and 4 are computing and components 3 and 5 are checkpointing. Then, the populations for classes 1 through 5 of the QN are 1, 1, 0, 1, 0 and the populations for classes 6 through 10 are 0, 0, 1, 0, 1.

The CPU and I/O service demands for classes $k = 1, \cdots, n$ are given by equations (2) and (3), respectively. The CPU and I/O service demands for class $n+k$ ($k = 1, \cdots, n$) are given by $C_{\text{CPU},k}$ and $C_{\text{I/O},k}$, respectively.

Appendix A provides the algorithm used to compute the transition rates $\mu_{i,j}$. These rates are used to put together the system of linear equations in Eqs. (24) and (25). Various numerical solvers can be used for that purpose. We used the solver in the R package that handles systems of linear equations with sparse matrices. The following additional notation is used in the remaining expressions.

- $v_k(b)$: value of the bit at position k (counting from the left) of binary number b. So, $v_4(000100) = 1$.
- $X_k(i)$: throughput of class k ($k = 1, \cdots, n$) for state i.
- $X_{n+k}(i)$: throughput of class $n+k$ ($k = 1, \cdots, n$) for state i.

With the values of p_i's we can obtain the probability P_k^c that component k is in the computing mode as the sum of all state probabilities where component k is computing. Note that P_k^c is also the average number, $\bar{n}_{c,k}$, of components of class k in the computing mode. Thus,

$$P_k^c = \bar{n}_{c,k} = \sum_{i=0}^{N-1} v_k(b(i))\ p_i. \tag{27}$$

The probability P_k^x that component k is checkpointing is simply $1 - P_k^c$ and is also equal to $\bar{n}_{x,k}$, the average number of components of class k in the checkpointing mode. Thus,

$$P_k^x = \bar{n}_{x,k} = 1 - P_k^c. \tag{28}$$

The average throughput, $\bar{X}_k$, of component k can be computed as

$$\bar{X}_k = \sum_{i=0}^{N-1} X_k(i)\ p_i. \tag{29}$$

Using the example of Fig. 2, we have that $\bar{X}_1 = X_1(3)p_3 + X_1(2)p2$. The overall average component throughput, $\bar{X}_{\text{COMP}}$, is

$$\bar{X}_{\text{COMP}} = \sum_{k=1}^{n} \bar{X}_k.P_k^c. \tag{30}$$

The average throughput, $\bar{X}_{n+k}$, of the checkpointing process for component k can be computed as

$$\bar{X}_{n+k} = \sum_{i=0}^{N-1} X_{n+k}(i)\ p_i. \tag{31}$$

In the example of Fig. 2, $\bar{X}_3 = X_3(0)p_0 + X_3(1)p_1$. The overall average throughput, $\bar{X}_{\text{CHECK}}$, of components in the checkpointing mode is

$$\bar{X}_{\text{CHECK}} = \sum_{k=1}^{n} \bar{X}_{n+k}.P_k^x. \tag{32}$$

According to Little's Law [18], the average time, $\bar{r}_{c,k}$, it takes component k to complete its computation between consecutive checkpoints is

$$\bar{r}_{c,k} = \frac{\bar{n}_{c,k}}{\bar{X}_k}. \quad (33)$$

Also according to Little's Law, the average time it takes a component k to complete its checkpoint is

$$\bar{r}_{x,k} = \frac{\bar{n}_{x,k}}{\bar{X}_{n+k}}. \quad (34)$$

The average total time, $\bar{R}_{c,k}$, it takes component k to complete its computation taking into account contention with other components and checkpointing is given by Equation (9) and the values of the overhead Ov_k, the availability A_k, and the relative progress RP_k are given by Equations (10)-(12), respectively.

When combining the individual availability of the components, one has to take into consideration how the system stakeholders view the relative importance of the different components. For example, the overall availability could be seen as the fraction of time that at least one component is available. In that case,

$$A = 1 - \prod_{k=1}^{n}(1 - A_k). \quad (35)$$

Alternatively, one could define the overall availability as a weighted average of the individual component availabilities:

$$A = \sum_{k=1}^{n} w_k.A_k \quad (36)$$

where $\sum_{k=1}^{n} w_k = 1$. Appendix B provides a discussion on the computational complexity and computation times of both models.

2. NUMERICAL EXAMPLES

We implemented both analytic models in Java and used some functions (e.g., solving systems of linear equations) provided by the R statistical package. We present here some numerical examples using the two models developed in the previous section. The time-related metrics presented in this section are normalized by the value of E, the average CPU time needed to complete the execution of a component. Thus, all normalized time-related metrics are dimensionless and have interesting interpretations. For example: R_c/E is the factor by which the average execution time of a component is elongated due to contention for resources, checkpointing, and rollback; T/E is the fraction of a component's total CPU time achieved between checkpoints; and $(1/\lambda)/E$ is the ratio between the mean time to fail and a component's total CPU time.

The top curve of Fig. 3 shows the variation of R_c/E with T/E for the case of a homogenous model with 3 components and Poisson failures with a normalized mean time to failure equal to 0.5 (i.e., two failures every execution of a component). The other normalized parameters are $E_{\text{CPU}} = 0.85, E_{\text{I/O}} = 0.15$, $C_{\text{CPU}} = 0.05$, $C_{\text{I/O}} = 0.075$, $RT_{\text{CPU}} = 0.06$ and $RT_{\text{I/O}} = 0.07$. The figure shows that R_c/E decreases with T, reaches a minimum, then starts to increase. This can be explained by looking at the change in the overhead, O_v, and the expected number of failures between consecutive checkpoints, NF, also shown on the same figure. The inflation in execution time for each component is caused by contention, failures and checkpointing overhead. The contention is constant in this example because the number of components is fixed. For smaller values of T, the checkpointing overhead is very high, however the number of expected failures in this small interval is small. After some point (the optimal T for minimum R_c), the decrease in O_v slows down while the number of failures increases fast. This increase has a bigger effect on R_c after this point.

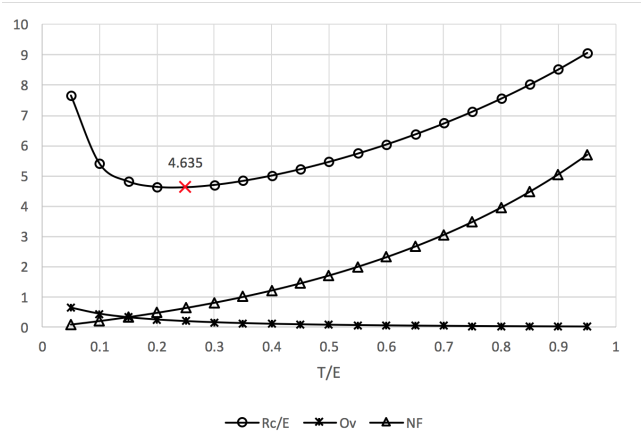

Figure 3: Normalized average execution time (R_c/E) vs. T/E for the homogeneous case and with Poisson failures. $(1/\lambda)/E = 0.5$**,** $n = 3$**.**

Figure 4 shows the normalized execution time vs. average availability for 1, 3, 5, and 7 concurrent components. The figure shows that as R_c decreases, A increases. Under constant contention, the achieved availability is affected by the same two factors as execution time: O_v (preventive maintenance), and NF (corrective maintenance). The figure shows that the optimal T value that minimizes execution time is usually the same or very close to the value that maximizes availability.

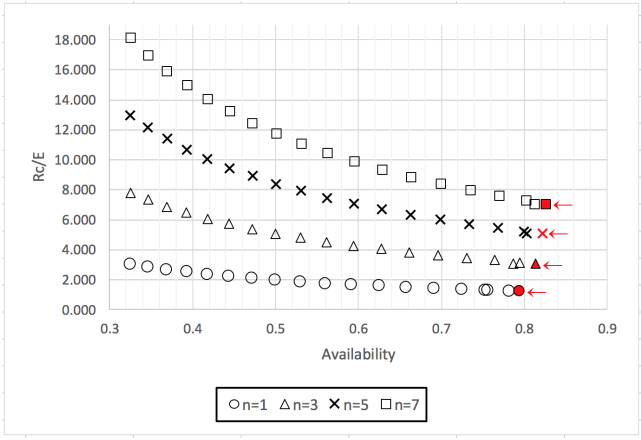

Figure 4: Normalized average execution time (R_c/E) vs. the availability A for the homogeneous case and with Poisson failures. $(1/\lambda)/E = 0.5$**,** $n = 1, 3, 5,$ **and 7.**

Figure 5 shows the variation in availability vs. T/E for

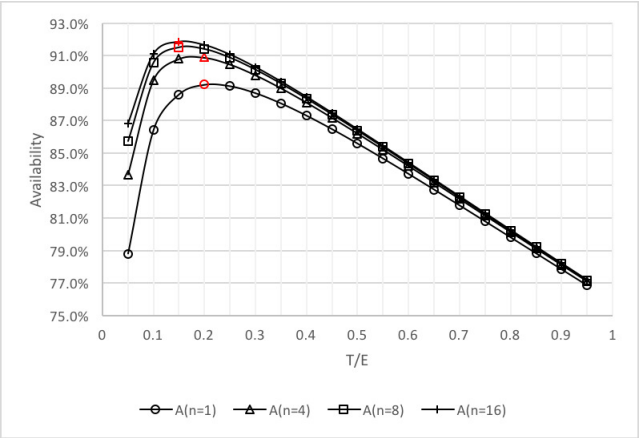

Figure 5: Availability A vs. T/E for $n = 1, 4, 8,$ and 16 for the homogeneous case and with Poisson failures. $(1/\lambda)/E = 2$.

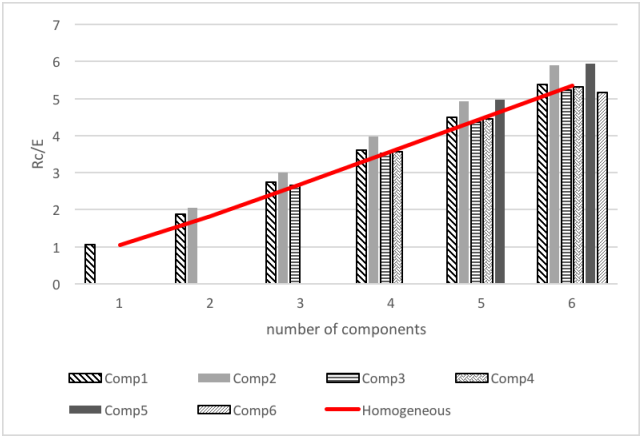

Figure 6: Normalized execution time (R_c/E) as components are incrementally added.

1, 4, 8 and 16 concurrent components and $\lambda = 1/(2E)$. In all four cases, the availability increases, reaches a maximum, and then decreases. This has the same explanation as Figs. 3 and 4. As R_c decreases, A increases, they reach their optimal values at very close values of T. After this point, R_c increases and A decreases.

Figure 6 illustrates the variation of the average normalized execution time as components are incrementally added from 1 to 6. The six components have different resource demands and different failure distributions. The characteristics of the six components vary as follows: $E = 15$ to 40 (with $E_{\text{CPU}} = 0.85E$), $T = 2$ to 4, $C_{\text{CPU}} = 0.015$ to 0.03, $C_{\text{I/O}} = 0.02$ to 0.04, $RT_{\text{CPU}} = 0.01$ to 0.028, and $RT_{\text{I/O}} = 0.002$ to 0.04. Components 1, 2, 4 and 5 have Poisson failures with failure rates equal to $0.03, 0.07, 0.02$,and 0.08, respectively. Components 3 and 6 have Weibull failure distributions with shape and scale parameters equal to (4, 50) and (3, 70), respectively. The figure also shows a red line that depicts how the normalized execution time would vary if all added components had the same characteristics as component 1. The figure also indicates how the mix of concurrent components influences on the normalized execution time. For example, R_c/E is very close to 6 for components 2 and 5 and it is close to 5 for all other components for $n = 6$.

3. EXPERIMENTAL VALIDATION

The first obvious validation of our models was to use the heterogeneous model in a situation in which the characteristics of all components were exactly the same. We then used the homogeneous model to obtain the metrics for the same inputs and obtained the same values provided by the heterogeneous model. This confirmed the correctness of our implementations of the two models. We had done unit tests previously to confirm that individual components of the implementation (e.g., QN model, Markov Chain solution) were correct. We then validated the models through both simulation and experiments as described in the next two subsections.

3.1 Validation Through Simulation

We wrote a Java program that simulates a node with a processor, a disk, and multiple software components. Each component is provided with a sequence of at least 500 jobs to execute, each representing a different execution of the component. The various properties of a component (e.g., execution time, checkpointing time, recovery time) are random variables with assigned distributions. For each task execution, values are drawn from the common distribution for that component. For example, the computation time of tasks for component k may be exponentially distributed with mean E_k, uniformly distributed checkpointing and roll back demands, and time to fail following a Weibull distribution with given shape and scale parameters. To ensure a constant level of contention during each run, the simulation run stops when one of the components finishes executing all of its tasks. At that point, the output results are averaged over all runs and 95% confidence intervals are computed over 40 runs in which each component is executed hundreds of times.. The analytic and simulation results are compared through the error $\epsilon = 100\times$ | simulation − analytic | /simulation.

Table 3 compares the execution time, availability and checkpoint overhead obtained by the analytic and simulation models for 5, 10, and 15 homogeneous components. The characteristics of the components are the same as component 1 in Table 4, which shows input data for a scenario with three different components that have different values of computing demands ($E_{\text{CPU},k}$ and $E_{\text{I/O},k}$), checkpointing demands ($C_{\text{CPU},k}$ and $C_{\text{I/O},k}$), and rollback time ($\text{RT}_{\text{CPU},k}$ and $\text{RT}_{\text{I/O},k}$).

We used the values in Table 4 as input parameters (means for exponential distributions) for the heterogeneous analytic model and the simulation for three different components. Table 7 displays several results for four versions of the experiments in which the value of T_k and the failure distribution ($\tilde{F}$) vary according to Table 5. In all variations, components 1 and 3 had exponentially distributed time to fail, while component's 2 time to fail followed a Weibull distribution. Both distributions have been used in many reliability studies. We used shape parameters of 4 and 4.5 for the Weibull distributions, which means that the failure rate increases over time.

Table 7 shows the average total execution time $R_{c,k}$, the availability, and the overhead for each component k under the analytic and simulation models along with 95% confidence intervals over 40 runs of the simulation in which each component executes to completion at least 500 times in each

Table 3: Comparison between analytic and simulation results for the homogeneous case

n	R_c			A			O_v		
	simulation	analytic	ϵ	simulation	analytic	ϵ	simulation	analytic	ϵ
5	3369.3 ± 32.5	3658.5	8.6%	$44.9\% \pm 0.2\%$	42.5%	5.4%	$27.3\% \pm 0.1\%$	28.0%	2.8%
10	6390.3 ± 47.9	7245.1	13.4%	$44.8\% \pm 0.1\%$	42.5%	5.1%	$27.4\% \pm 0.1\%$	28.1%	2.3%
15	9647.2 ± 61.4	10876	12.7%	$44.5\% \pm 0.1\%$	42.5%	4.4%	$27.7\% \pm 0.1\%$	28.0%	1.1%

Table 4: Input parameters for the results on Tables 7 and 8

k	$E_{\text{CPU},k}$	$E_{\text{I/O},k}$	$C_{\text{CPU},k}$	$C_{\text{I/O},k}$	$\text{RT}_{\text{CPU},k}$	$\text{RT}_{\text{I/O},k}$
1	303.75	119.50	25.40	9.99	30.20	11.88
2	151.25	59.50	18.70	7.36	23.30	9.17
3	203.75	106.88	48.00	18.88	24.70	9.72

Table 5: Four versions of the experiments for results on Tables 7 and 8

version	T	$\tilde{F}_1.\lambda$	$\tilde{F}_2.(shape, scale)$	$\tilde{F}_3.\lambda$
1	25	0.009	$(4.5, 115)$	0.013
2	25	0.008	$(4, 67)$	0.011
3	50	0.009	$(4.5, 115)$	0.013
4	50	0.008	$(4, 67)$	0.011

run. In each run, each component is given a new seed to generate random samples using the means in Table 4. Note that all three components are executing concurrently during the simulation runs and sharing the CPU and I/O resources. Table 7 shows that the error of the analytic model for $R_{c,k}$ is between 9.5% and 11.4%, for exponentially distributed failures. The errors for the Weibull distribution are higher, i.e., around 14.5%, but still acceptable for execution times when queuing effects are present [16]. The lower errors for the exponential distribution may be due to its memoryless property, which is consistent with the underlying Markovian assumptions of the models presented here. But, even under non-Markovian failure assumptions, our models exhibit fairly robust prediction capabilities.

3.2 Validation Through Experimentation

A second validation was done through experimentation. We developed a micro-benchmark in C that implements software components that perform jobs repeatedly, do frequent checkpoints and fail randomly according to an exponential or Weibull distribution. Components receive as input the number of jobs, the time between consecutive checkpoints (T), and the failure rate (λ). At the beginning of each job, the component starts a timer, resets the time to checkpoint to T, and generates the next failure time using λ. During the task's execution, a component constantly checks whether it is time to fail or checkpoint. If it is time to fail, the component rollbacks to the latest checkpoint and redoes the work that was not checkpointed. If it is time to checkpoint, the components does so. After each failure or checkpoint, the component generates a new failure time, resets the time to checkpoint to T, and resets the un-checkpointed work to zero. At the end of a job, the component stops the timer and records the time taken to complete the task ($\bar{R}_c$ in our model).

Each component alternates between CPU and I/O activity by invoking the *compute*() and *doIO*() procedures (see Algorithm 1). The first performs thousands of calls to trigonometric functions and stores the results in an array. The second writes the array to a binary file. These two procedures were run thousands of times to measure the CPU and disk service demands. The CPU demand was measured using C's `clock()` function, which returns the number of clock ticks since the program was started. This value is then divided by the global variable `CLOCKS_PER_SEC` to obtain the time spent using the CPU. The disk demand was measured by running the `fs_usage` command to measure the time needed for all file system calls. Table 6 shows the CPU and disk demands of the functions *compute*() and *doIO*() averaged after 2000 runs each with 95% confidence intervals. All experiments were run on a MacBook Pro with a 2.6 GHz Intel Core i5 processor. Only one core was enabled to produce contention during experimentation with multiple concurrent components. The pseudo code for the micro-benchmark is given in Algorithms 1, 2, and 3.

Algorithm 1 Micro-benchmark Pseudo Code

```
    Input: numTasks, iterations, numCompute, numIO, λ, T
    for t = 1 → numTasks do
      Reset();
      START ← current_time();
 5:   for i = 1 → iterations do
        for j = 1 → numCompute do
          Execute(compute(), compute_time, uncheckedComp);
        end for
        for j = 1 → numIO do
10:       Execute(doIO(), IO_time, uncheckedIO);
        end for
      end for
      END ← current_time();
      taskTimes[t] ← END − START
15: end for
    printTaskTimes();
```

Table 6: *compute*() **and** *doIO*() **CPU and disk demands in msec with 95% confidence intervals**

	CPU demand	disk demand
compute()	2.66 ± 0.008	0
doIO()	0.666 ± 0.005	0.0873 ± 0.04

Similarly to the simulation, four experiments were conducted according to Tables 4 and 5. For each experiment, 40 runs were conducted. In each run, each of the concurrent components executed at least 50 jobs. The values generated using the means in Tables 4 and 5 were averaged and used as an input to the analytical model. The total execution time,

Algorithm 2 Execute Pseudo Code

```
    Input: procedure, duration, counter
    call(procedure);
    timeToFail = timeToFail − duration;
    timeToCheck = timeToCheck − duration;
 5: if timeToFail ≤ 0 then
      Rollback();
      Redo(uncheckedComp, uncheckedIO);
      Reset();
    else if timeToCheck ≤ 0 then
10:   Checkpoint();
      Reset();
    else
      counter ← counter + 1;
    end if
```

Algorithm 3 Reset Pseudo Code

```
timeToCheck ← T
timeToFail ← nextExponential(λ)
uncheckedComp ← 0
uncheckedIO ← 0
```

availability and checkpoint overhead were averaged over the jobs in each run, then over all the runs and compared to the results from the analytical model (see Table 8).

4. RELATED WORK

None of the prior work mentioned below considers, as we do, the effect of resource contention with other components or with checkpointing processes. There is a wealth of publications on analytic modeling of rollback and checkpointing since the work of Young in 1974 [26]. We highlight a few here (see [25] for an extensive bibliography). Gelenbe and colleagues developed comprehensive models for rollback and checkpointing under various assumptions regarding failure time distribution and static versus dynamic checkpointing [11, 13, 12]. Other analytic models can be found in Chandy et. al. [2] and Tantawi and Ruschitzka [23].

Nicola and Spanje [21] study and compare different checkpointing strategies and models in order to select one that adequately represents a realistic system and is yet tractable for analysis. Dimitrov et al. [6] developed analytic models to find a checkpointing schedule that optimizes a job's total processing time under implicit breakdowns, i.e., failures are not detected immediately but a special test has to be performed to detect the failure.

Kishor Trivedi has done substantial work in using performance modeling to assess software reliability and the impacts of software rejuvenation [10]. The work in [17] by Ling et al. uses variational calculus to derive a closed form expression for the optimal checkpointing frequency as a function of the failure rate with the goal of minimizing the total expected cost of checkpointing and recovery.

Daly [4] provides a high order estimate of the optimum checkpoint interval to minimize total application runtime under Poisson failures. Chen and Ren [3] analyze the relationships between checkpoint interval and system availability, task execution time, and task deadline miss probability, for soft real-time applications. Bougeret et al. [1] develop solutions for optimal checkpointing that minimize execution time for sequential and parallel jobs with Poisson failures and use a dynamic programming heuristic for the case of Weibull failures.

Lu et al. [19] derive optimal checkpointing intervals for systems with latent errors, i.e., errors that may go undetected for some time. This assumption is more realistic than that of immediate failure detection assumed by the vast majority of the checkpointing modeling work, including ours. The authors discuss the importance of multiversion checkpoints to achieve acceptable failure coverage. Di et al. [5] present a sophisticated deterministic multilevel checkpoint optimization model in the context of exascale systems with a large number of multi-core nodes. The authors consider a parallel application with many processes running on many cores. Jones et al. [14] use simulation with real workload data to demonstrate the impact of sub-optimal checkpoint intervals on application efficiency in HPC clusters. No analytic model is presented. A comprehensive survey of roll-back recovery protocols in message-passing systems was presented in [7]. Recent studies have leveraged the use of NVRAM as a replacement to disk to store checkpoints. Gao et al [9].

5. CONCLUDING REMARKS

This paper presented two analytic models for assessing the performance of software systems that use rollback and checkpointing to improve their reliability. Differently from previous work, our models allow for a compute node to run many different software components concurrently, each possibly having different resource demand and failure distribution characteristics. Our analytic models compute the average execution time per component, their availability, overhead due to checkpointing and rollback, and relative progress, while taking into account the impact of contention for shared resources such as processors and I/O devices. Simulation and experimental results used to validate the models showed very high accuracy for Poisson failures. Simulation results showed acceptable accuracy for Weibull failures. In the future, we intend to integrate our models into an autonomic controller that can dynamically tweak the various parameters of the model to maximize a utility function of execution time, availability, and overhead.

Acknowledgements

This work was partially supported by the AFOSR grant FA9550-16-1-0030.

6. REFERENCES

[1] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. Checkpointing strategies for parallel jobs. In *2011 Intl. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. IEEE, 2011.

[2] K. M. Chandy, J. C. Browne, C. W. Dissly, and W. R. Uhrig. Analytic models for rollback and recovery strategies in data base systems. *IEEE Tr. Software Engineering*, (1):100–110, 1975.

[3] N. Chen and S. Ren. Adaptive optimal checkpoint interval and its impact on system's overall quality in soft real-time applications. In *Proc. 2009 ACM Symp. Applied Computing*, pages 1015–1020. ACM, 2009.

Table 7: Comparison between analytic and simulation results for the heterogeneous case.

Experiment	k	R_c			A			O_v		
		simulation	analytic	ϵ	simulation	analytic	ϵ	simulation	analytic	ϵ
1	1	2525.8 ± 30.7	2766.7	9.5%	$36.6 \pm 0.1\%$	35.1%	4.1%	$46.1 \pm 0.1\%$	46.8%	1.5%
	2	857.1 ± 5.4	960.9	12.1%	$54.3 \pm 0.1\%$	50.2%	7.5%	$45.6 \pm 0.1\%$	49.7%	8.9%
	3	2712.5 ± 31.0	3009.2	10.9%	$26.0 \pm 0.1\%$	23.0%	11.3%	$58.2 \pm 0.1\%$	61.1%	5.0%
2	1	2469.9 ± 20.3	2704.1	9.5%	$37.3 \pm 0.1\%$	35.9%	3.8%	$47.2 \pm 0.1\%$	47.9%	1.6%
	2	874.5 ± 4.5	979.0	11.9%	$53.5 \pm 0.1\%$	49.4%	7.6%	$44.9 \pm 0.1\%$	48.6%	8.2%
	3	2648.8 ± 24.3	2922.5	10.3%	$26.8 \pm 0.1\%$	23.9%	10.9%	$59.7 \pm 0.2\%$	62.7%	5.0%
3	1	2078.2 ± 22.8	2295.2	10.4%	$45.9 \pm 0.2\%$	42.4%	7.8%	$26.9 \pm 0.1\%$	28.1%	4.4%
	2	642.0 ± 3.2	735.1	14.5%	$72.3 \pm 0.1\%$	65.5%	9.4%	$26.2 \pm 0.1\%$	32.3%	23.0%
	3	2074.6 ± 26.3	2310.6	11.4%	$35.9 \pm 0.2\%$	30.0%	16.5%	$35.5 \pm 0.2\%$	39.8%	12.3%
4	1	2008.0 ± 17.9	2213.7	10.2%	$47.7 \pm 0.2\%$	44.1%	7.4%	$27.9 \pm 0.1\%$	29.3%	4.8%
	2	839.2 ± 5.6	959.8	14.4%	$60.3 \pm 0.1\%$	50.3%	16.6%	$21.6 \pm 0.1\%$	24.9%	14.9%
	3	1976.9 ± 21.7	2186.7	10.6%	$37.7 \pm 0.2\%$	32.1%	14.8%	$37.9 \pm 0.2\%$	42.2%	11.4%

Table 8: Comparison between analytic and experimentation results for the heterogeneous case.

Experiment	k	R_c			A			O_v		
		experiment	analytic	ϵ	experiment	analytic	ϵ	experiment	analytic	ϵ
1	1	2688.46 ± 88.30	2727.87	1.5%	$37.5\% \pm 0.7\%$	34.3%	8.6%	$45.8\% \pm 1.0\%$	47.2%	3.2%
	2	1014.58 ± 31.51	956.58	5.7%	$51.7\% \pm 0.5\%$	50.4%	2.6%	$48.1\% \pm 0.5\%$	49.5%	2.8%
	3	2730.61 ± 156.16	2763.84	1.2%	$25.5\% \pm 0.6\%$	24.1%	5.6%	$58.8\% \pm 0.9\%$	60.3%	2.5%
2	1	2798.21 ± 139.66	2858.18	2.1%	$37.2\% \pm 0.5\%$	35.7%	4.1%	$47.6\% \pm 0.8\%$	48.0%	0.8%
	2	1011.36 ± 24.87	984.72	2.6%	$51.0\% \pm 0.6\%$	49.6%	2.7%	$47.7\% \pm 0.6\%$	48.6%	2.0%
	3	2736.92 ± 102.02	2929.27	7.0%	$25.8\% \pm 0.5\%$	23.6%	8.6%	$60.2\% \pm 0.7\%$	62.9%	4.5%
3	1	2167.65 ± 93.79	2309.34	6.5%	$47.4\% \pm 1.0\%$	42.4%	10.5%	$25.4\% \pm 0.8\%$	28.0%	10.4%
	2	754.13 ± 25.89	741.69	1.6%	$69.7\% \pm 0.7\%$	66.0%	5.2%	$28.9\% \pm 0.6\%$	31.9%	10.3%
	3	1988.79 ± 65.21	2354.04	18.4%	$38.0\% \pm 0.9\%$	30.6%	19.6%	$33.2\% \pm 1.2\%$	39.5%	18.7%
4	1	2255.41 ± 206.68	2225.69	1.3%	$48.9\% \pm 0.9\%$	44.4%	9.1%	$26.8\% \pm 1.0\%$	29.1%	8.7%
	2	924.12 ± 63.52	967.16	4.7%	$59.9\% \pm 0.7\%$	50.3%	16.0%	$21.5\% \pm 0.6\%$	24.6%	14.3%
	3	1943.90 ± 154.25	2171.24	11.7%	$39.4\% \pm 1.0\%$	31.9%	18.8%	$36.1\% \pm 0.8\%$	42.3%	17.1%

[4] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

[5] S. Di, L. Bautista-Gomez, and F. Cappello. Optimization of a multilevel checkpoint model with uncertain execution scales. In *Proc. Intl. Conf. High Performance Computing, Networking, Storage and Analysis*, pages 907–918. IEEE Press, 2014.

[6] B. Dimitrov, Z. Khalil, N. Kolev, and P. Petrov. On the optimal total processing time using checkpoints. *IEEE Tr. Software Engineering*, 17(5):436, 1991.

[7] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[8] E. Elsayed. *Reliability Engineering*. Number v. 1 in Reliability Engineering. Addison Wesley Longman, 1996.

[9] S. Gao, B. He, and J. Xu. Real-time in-memory checkpointing for future hybrid memory systems. In *Proc. 29th ACM on Intl. Conf. Supercomputing*, pages 263–272. ACM, 2015.

[10] S. Garg, Y. Huang, C. Kintala, and K. S. Trivedi. Minimizing completion time of a program by checkpointing and rejuvenation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 24, pages 252–261. ACM, 1996.

[11] E. Gelenbe. A model of roll-back recovery with multiple checkpoints. In *Proc. 2nd Intl. Conf. Software Engineering*, pages 251–255. IEEE Computer Society Press, 1976.

[12] E. Gelenbe. On the optimum checkpoint interval. *J. ACM*, 26(2):259–270, 1979.

[13] E. Gelenbe and D. Derochette. Performance of rollback recovery systems under intermittent failures. *C. ACM*, 21(6):493–499, 1978.

[14] W. M. Jones, J. T. Daly, and N. DeBardeleben. Application monitoring and checkpointing in hpc: looking towards exascale systems. In *Proc. 50th Annual Southeast Regional Conference*, pages 262–267. ACM, 2012.

[15] L. Kleinrock. *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975.

[16] E. Lazowska, J. Zahorjan, G. Graham, and K. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice Hall, 1984.

[17] Y. Ling, J. Mi, and X. Lin. A variational calculus approach to optimal checkpoint placement. *IEEE Tr. Computers*, 50(7):699–708, 2001.

[18] J. D. Little. A proof for the queuing formula: L= λ w. *Operations Research*, 9(3):383–387, 1961.

[19] G. Lu, Z. Zheng, and A. A. Chien. When is multi-version checkpointing needed? In *Proc. 3rd Wkhp. Fault-tolerance for HPC at extreme scale*, pages 49–56. ACM, 2013.

[20] D. A. Menascé, V. A. Almeida, L. W. Dowdy, and L. Dowdy. *Performance by design: computer capacity planning by example*. Prentice Hall Professional, 2004.

[21] V. F. Nicola and J. M. Van Spanje. Comparative analysis of different models of checkpointing and recovery. *IEEE Tr. Software Engineering*, 16(8):807–821, 1990.

[22] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2016.

[23] A. N. Tantawi and M. Ruschitzka. Performance analysis of checkpointing strategies. *ACM Tr. Computer Systems (TOCS)*, 2(2):123–144, 1984.

[24] S. Urbanek. *Rserve: Binary R server*, 2013. R package version 1.7-3.

[25] K. Wolter. *Stochastic Models for Fault Tolerance, Restart, Rejuvenation, and Checkpointing*. Springer Verlag, 2010.

[26] J. W. Young. A first order approximation to the optimum checkpoint interval. *C. ACM*, 17(9):530–531, 1974.

[27] W. Zhao. *Building dependable distributed systems*. John Wiley & Sons, 2014.

APPENDIX

A. TRANSITION RATES FOR THE HETEROGENEOUS MODEL

The following notation is used in Algorithm 4 that describes how the transition rates $\mu_{i,j}$ for $i = 0, \cdots, N-1$ and $j = 0, \cdots, N-1$ are computed.

- $n_c(i)$: number of components in the computing mode in state i. This is equal to the number of 1's in $b(i)$.
- $n_x(i)$: number of components in checkpointing mode in state i. This is equal to the number of 0's in $b(i)$.
- $f(b)$: position of the first bit 1 in a binary number b with a single 1 bit. These positions are counted from the left to the right starting at 1. So, $f(000100) = 4$. $f(b)$ can be computed as $n - \log_2 b_{10}$ where n is the number of bits in b and b_{10} is the decimal value that corresponds to b.
- $v_k(b)$: value of the bit at position k (counting from the left) of binary number b. So, $v_4(000100) = 1$.
- $n1(b)$: number of bits equal to 1 in the binary number b.
- $X_k(i)$: throughput of class k ($k = 1, \cdots, n$) for state i.
- $X_{n+k}(i)$: throughput of class $n+k$ ($k = 1, \cdots, n$) for state i.

B. MODEL COMPUTATIONAL EFFORT

The homogeneous model requires solving a 2-class closed MVA model $n+1$ times for populations $(0, n), (1, n-1), \cdots, (n, 0)$. We used exact MVA [20] but we substantially reduced the computational complexity of solving the $n+1$ QNs by storing in a hash table for further reuse the results for subpopulations as they are obtained. As Fig. 7 indicates, the traditional approach requires close to 1,000 evaluations of a QN model for 16 components while the optimized method requires around 160 evaluations.

Algorithm 4 Computation of $\mu_{i,j}$'s

```
Input: N
for i = 0 → N − 1 do
  for j = 0 → N − 1 do
    if i=j then
      μ_{i,j} = 0
    else
      /* find the component k that changed mode. */
      /* check different bits between states i and j */
      bs = b(i) ⊕ b(j);
      if n1(bs) = 1 then
        /* There is only one bit of difference */
        k = f(bs);
        if v_k(b(i)) = 1 then
          μ_{i,j} = X_k(i) /* comp. k starts checkpointing */
        else
          μ_{i,j} = X_{n+k}(i) /* comp. k finished checkpointing */
        end if
      else
        μ_{i,j} = 0 /* more than one bit of difference */
      end if
    end if
  end for
end for
```

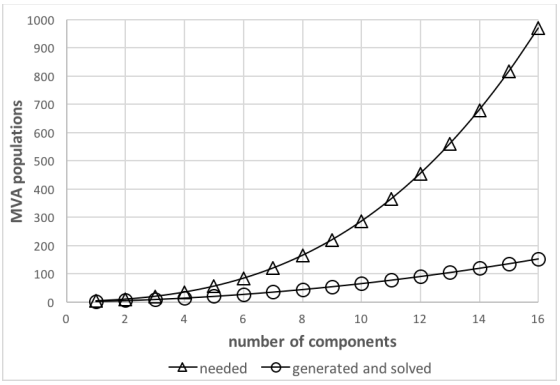

Figure 7: Number of times a QN needs to be evaluated for the homogeneous model under traditional methods and under our approach.

For the heterogeneous model, all n transition rates out of a given state can be obtained by solving a single multiclass QN. We use approximate MVA (AMVA) [20] instead of exact MVA to speed up the model solution. We need to solve 2^n AMVA QNs with $2n$ classes. But the population of these classes is limited to 1, which reduces the computational complexity of solving each of these QNs.

The heterogeneous model requires the solution of the system of linear equations in Eqs. (24) and (25). For a system of n components, the system of linear equations is a $2^n \times 2^n$ sparse matrix that has $(2^n - 1)(n + 1) + 2^n$ non-zero elements. We leveraged the sparseMatrix library in R [22] by sending requests to an instance of Rserve [24] running on the same machine. It took about 3.5 minutes to solve a system with 13 components on a laptop with a 2.6 GHz Intel Core i5 processor and 8 GB 1600 MHz DDR3.