applied sciences

MDPI

Article

An Efficient Library for Reliability Block Diagram Evaluation

Laura Carnevali, Lorenzo Ciani, Alessandro Fantechi, Gloria Gori * and Marco Papini

Department of Information Engineering (DINFO), School of Engineering, University of Florence, via di S. Marta 3, 50139 Florence, Italy; laura.carnevali@unifi.it (L.C.); lorenzo.ciani@unifi.it (L.C.); alessandro.fantechi@unifi.it (A.F.); marco.papini@unifi.it (M.P.)
* Correspondence: gloria.gori@unifi.it

Abstract: Reliability Block Diagrams (RBDs) are widely used in reliability engineering to model how the system reliability depends on the reliability of components or subsystems. In this paper, we present *librbd*, a C library providing a generic, efficient and open-source solution for time-dependent reliability evaluation of RBDs. The library has been developed as a part of a project for reliability evaluation of complex systems through a layered approach, combining different modeling formalisms and solution techniques at different system levels. The library achieves accuracy and efficiency comparable to, and mostly better than, those of other well-established tools, and it is well designed so that it can be easily used by other libraries and tools.

Keywords: Reliability Block Diagrams (RBD); hierarchical reliability model; reliability curve; reliability evaluation; software libraries

Citation: Carnevali, L.; Ciani, L.; Fantechi, A.; Gori, G.; Papini, M. An Efficient Library for Reliability Block Diagram Evaluation. *Appl. Sci.* **2021**, *11*, 4026. https://doi.org/10.3390/app11094026

Academic Editors: Andrea Bondavalli and Andrea Ceccarelli

Received: 30 March 2021
Accepted: 22 April 2021
Published: 28 April 2021

1. Introduction

Reliability is defined as "the ability of a system or component to perform its required functions under stated conditions for a specified period of time" [1]. Reliability is often expressed through the usage of probability theory, i.e., it is defined as the probability that the system has successfully performed its required functions in time interval $[t_0, t)$ given that it was correctly operating at time t_0 [2].

Reliability of a complex system is assessed by using a reliability model. Several reliability models have been developed. These models can be divided into the following two main categories:

- **Combinatorial models**: they allow to efficiently evaluate reliability under the strong assumption of statistically independent components [3,4]. These models include Reliability Block Diagrams (RBDs) [5,6], Fault Trees (FTs) [7,8], Reliability Graphs (RGs) [9,10] and Fault Trees with Repeated Events (FTREs) [8,11].
- **State-space based models**: they allow for the modeling of several dependencies among failures, including statistical, time and space dependency, at the cost of a difficult tractability due to the state-space explosion [3,4]. These models include Continuous Time Markov Chains (CTMCs) [12,13], Stochastic Petri Nets (SPNs), Generalized Stochastic Petri Nets (GSPNs) and Stochastic Time Petri Nets (STPNs) [14–17], Stochastic Reward Nets (SRNs) [18,19] and Stochastic Activity Networks (SANs) [20,21].

The expressive power of state-space based models is obviously greater than the one of combinatorial models. On the other hand, expressive power varies among the different combinatorial models [22].

All models that exploit both the usage of combinatorial and state-space based solutions for the quantitative evaluation are classified as hybrid models and are considered as the state-of-the-art approach to dependability evaluation [3]. Both Dynamic RBD (DRBD) [23,24] and Dynamic FT (DFT) [25–27] are hybrid models, since they combine CTMC [13] evaluation with, respectively, RBD [6] and FT [8] quantitative analysis. Hierarchical models, i.e., models that combine the usage of different formalisms in order to

analyze the system at different levels, have been proposed in order to both exploit the benefits and to limit the drawbacks of combinatorial and state-space based models [3,28,29].

This paper is structured as follows: Section 1 presents the context and motivation for this work; Section 2 recalls the definition of RBDs and the mathematics used to evaluate them; Section 3 describes the design and optimizations of the implemented RBD computation library; Section 4 presents the materials and the methods used to obtain the results; Section 5 evaluates the performance of the RBD computation library and discusses the results; finally, Section 6 concludes the paper with some final remarks.

Context and Motivation

Our aim is to support the layered approach presented in [28,29], where RBDs are adopted to model major transitions of system structure (e.g., in a reconfiguration), while the finer modeling of the lower levels is based on STPNs and GSPNs. Our goal consists of the definition of a predictive diagnostics approach for the health assessment of complex systems. Specifically, we propose the usage of diagnostics data to estimate the reliability of basic components, leveraging the usage of a reliability hierarchical model to estimate the reliability curve of the system under analysis. By using this *tuned* reliability curve, we can compute the probability of system failure in a given future time interval, thus implementing a predictive diagnostics system. This approach requires a frequent evaluation of the reliability curve, hence efficient tools to evaluate it are needed.

Consider, for example, the system shown in Figure 1. The system has been subdivided into four statistically independent subsystems. The subsystems C_1 and C_2 model two identical power supplies in current sharing, C_3 is the computing subsystem, while C_4 is the acquisition subsystem. Each separate subsystem can then be modeled using an STPN/GSPN: in this example, all subsystems are modeled using GSPNs.

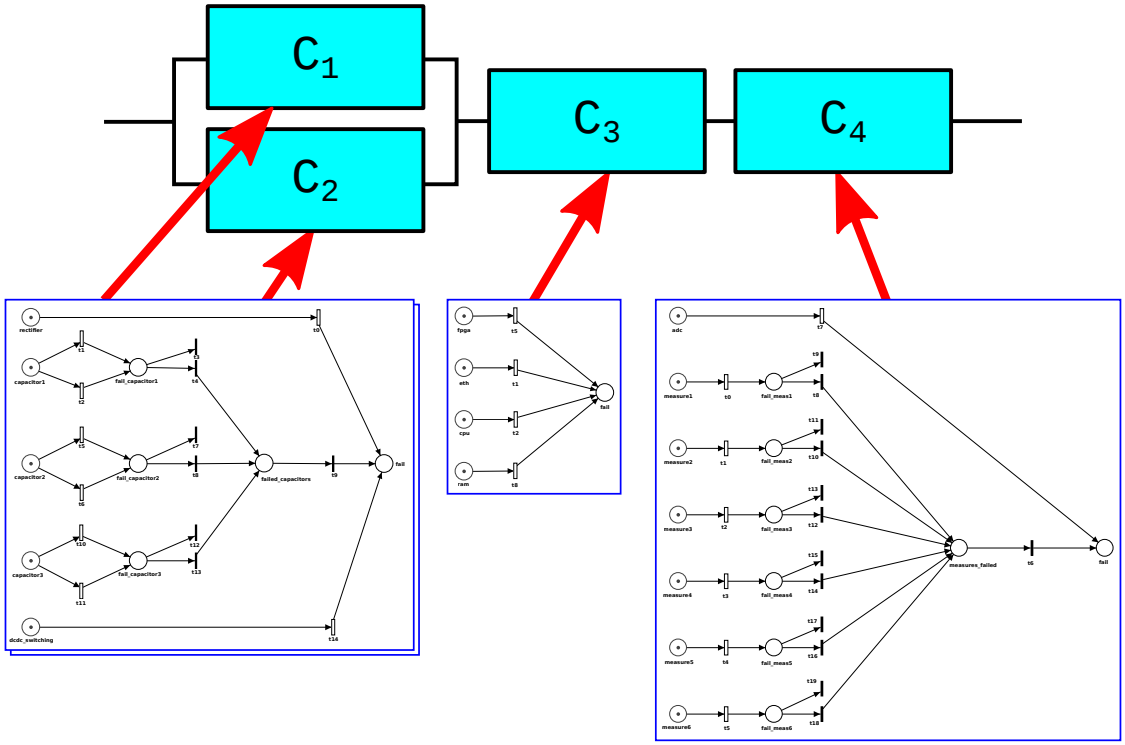


Figure 1. Example of layered reliability model.

The approach to the reliability analysis using this layered model is shown in Figure 2. The input data of this approach is the failure rate function $\lambda(t)$ for each modeled component. By inserting the failure rates into the STPN/GSPN models, we can analyze the models and we obtain, for each modeled subsystem, its reliability curve. Please note that, by modifying

the failure rate function $\lambda(t)$ of at least one modeled component, we have an impact on its distribution of failures and, as a consequence, we produce a reliability curve of the modeled subsystem with a different shape. Furthermore, by varying both the failure rate functions and their parameters, it is possible to refine the model, hence considering the uncertainties of the model.

Finally, by feeding the reliability curves of all subsystems into the RBD model, we can analyze the whole system and we obtain its reliability curve. Please note that the input data of this second phase are the reliability curves of all subsystems obtained through the analysis of the STPN/GSPN models. Thus, we combine the strength of combinatorial approaches, i.e., their efficiency, with the one of state-space based ones, i.e., their ability to model the statistical dependence of faults.

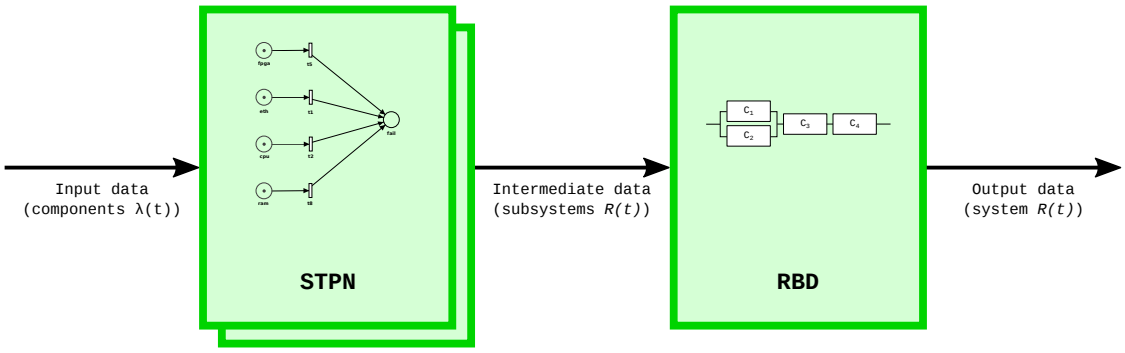


Figure 2. Application of layered reliability evaluation.

This hierarchical approach to the reliability evaluation can be extended to model the system-level reliability, i.e., the reliability of the whole system composed by both hardware and software. In general, hardware-related failures are statistically independent from software-related ones, i.e., software bugs [30,31]. The estimation of the number of failures in the source code is a difficult task [32]. However, in recent years, several methodologies have been developed to model and increase the software reliability [33–35].

The ORIS tool [36] has been adopted to support STPN and GSPN modeling. In this paper, we focus on the implementation of an efficient tool to evaluate RBD blocks.

More specifically, we looked for a tool with the following characteristics:

- To be highly optimized.
- To allow the resolution of RBD basic blocks (excluding singleton given its trivial formula).
- To allow the reliability computation of an RBD basic block in a time interval.
- To be available as a free software.
- To be available for the most common Operating Systems (OS), i.e., Windows, Mac OS and Linux.

Several tools for RBD definition and analysis exist, although the majority of them are commercial tools. We provide a list of tools that were considered during our work:

- **RBDTool**: this open-source and multiplatform tool allows the definition of RBD models and it provides support for their quantitative analysis [37].
- **Edraw Block Diagram**: this commercial tool allows the definition of RBD models [38].
- **Reliability Workbench**: this commercial tool allows the definition and analysis of scalable RBD models through the usage of submodels. Furthermore, it supports the *minimal cut-set* analysis of the RBD model [39].
- **Relyence RBD**: this commercial tool has features comparable with Reliability Workbench [40].
- **SHARPE**: the Symbolic Hierarchical Automated Reliability and Performance Evaluator (SHARPE) tool is a general hierarchical modeling tool that analyzes stochastic models of reliability, availability, performance and performability [41,42]. This tool al-

lows the definition of hierarchical reliability models with several formalisms, including RBDs, and it supports the time-dependent reliability analysis.

Of all considered tools, SHARPE is the closest to all our requirements. Since no one fits completely all of them, we have implemented a custom library that provides the RBD evaluation, from now on referred as *librbd*. More specifically, the *librbd* library supports the numerical computation of the reliability curve for all RBD basic blocks, it exploits several optimizations and multithreading paradigm, and it is multiplatform. Finally, we have publicly released this open source library under the AGPL v3.0 license [43].

2. Reliability Block Diagrams

An RBD decomposes a system into its independent components and shows the logical connections needed for the successful operation of the system [3–5,44,45]. The basic assumptions of the RBD methodology are the following ones:

1. The modeled system, as well as each component, has only two states, i.e., *success* and *failure*.
2. The RBD represents the *success* state of the modeled system through the usage of *success paths*, i.e., the connections of the success states of its components.
3. The system components are statistically independent. Under this assumption, the probability of failure of the block A, $P(A)$, is not related with the probability of failure $P(B)$ of the block $B \; \forall \; A, B$ such that $A \neq B$.

$$P(A|B) = P(A) \; \forall \; A, B \mid A \neq B \tag{1}$$

2.1. Basic Blocks

An RBD is built by drawing success paths between blocks composing the system. In order to correctly model an RBD, the following basic blocks are defined:

- **Singleton**. This block is the simplest one and it is composed by a single component. The block state is equal to *success* if and only if the component is in *success* state. An example is a stand-alone Power Supply.
- **Series**. This block is composed by N components. The block state is equal to *success* if and only if all the components are in *success* state. An example is a 2-out-of-2 computing system (2oo2).
- **Parallel**. This block is composed by N components. The block state is equal to *success* if and only if at least one component is in *success* state, or, in other terms, the block state is equal to *failure* if and only if all the components are in the *failure* state. An example is a redundant Power Supply system with current sharing.
- ***K*-out-of-*N* (*KooN*)**. This block is composed by N components. The block state is equal to *success* if and only if at least K components out of N are in *success* state. An example is a 2-out-of-3 computing system (2oo3).
- **Bridge**. This block is composed by 5 components arranged as shown in Figure 3. The block state is equal to *success* if at least one of the four following conditions is satisfied:
  1. Components A and B are correctly operating.
  2. Components C and D are correctly operating.
  3. Components A, E and D are correctly operating.
  4. Components C, E and B are correctly operating.

  An example is a network infrastructure.

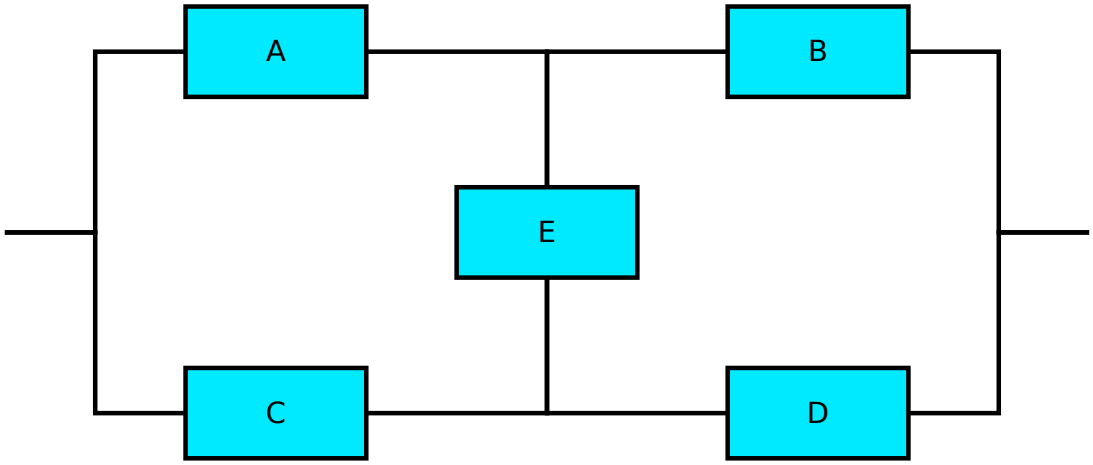


Figure 3. Layout of RBD bridge block.

2.2. Quantitative Evaluation Using RBDs

In this section, we recall the mathematical formulas used to quantitatively evaluate the probability that a block is correctly operating, i.e., its state is equal to *success*, by using the RBD formalism. More specifically, we firstly introduce the general formulas that can be always used; afterwards we present simplified formulas that can be used if and only if all components inside a block are equal, i.e., they have the same probability of being in *success* state.

Let p_i be the probability that the state of the i-th component is equal to *success*, and let $q_i = 1 - p_i$ be the probability that the state of the i-th component is equal to *failure*.

Since a singleton block is composed by only one component, its probability of being in *success* state $p_{singleton}$ is trivially equal to the probability of being in *success* state of its sole component p.

2.2.1. Quantitative Evaluation: General Formulas

The following general formulas can be used to quantitatively evaluate the probability that the state of an RBD block composed by N components is equal to *success*:

- **Series**. The probability of *success* of the series block p_{series} is computed as:

$$p_{series} = \prod_{i=1}^{N} p_i \tag{2}$$

- **Parallel**. The probability of *failure* of the parallel block $q_{parallel}$ is computed as:

$$q_{parallel} = \prod_{i=1}^{N} q_i = \prod_{i=1}^{N} (1 - q_i) \tag{3}$$

The probability of *success* of the parallel block $p_{parallel}$ is thus computed as:

$$p_{parallel} = 1 - \prod_{i=1}^{N} (1 - p_i) \tag{4}$$

- ***K*-out-of-*N* (*KooN*)**. In order to compute the probability of *success* of a *KooN* block, we can use one of the following approaches:

  1. Let $C(N, i, j)$ be the j-th unique combination of i out of N components correctly working. For a given couple $<i, N>$, the number of unique combinations is equal to the binomial coefficient $\binom{N}{i}$. We define $path(N, i, j)$ as the specific realization of one of the possible system states for which i components out of N are correctly working while the other $(N - i)$ have failed: the exact set of

the working components is selected through the usage of unique combination $C(N,i,j)$. Its probability of occurrence is:

$$P_{path(N,i,j)} = \prod_{l \in C(N,i,j)} p_l \cdot \prod_{m \notin C(N,i,j)} q_m \tag{5}$$

The state of a *K*oo*N* block is equal to *success* if and only if the current system state is satisfied by one path of at least K working components. The probability of *success* of the *K*oo*N* block can be defined as:

$$p_{KooN} = \sum_{i=K}^{N} \sum_{j=1}^{\binom{N}{i}} P_{path(N,i,j)} \tag{6}$$

2. Observe that the probability of *success* of a system with 0 or more components out of I in *success* state is equal to 1 and observe that the probability of *success* of a system with J or more components out of I with $J > I$ in *success* state is equal to 0. A recursive approach for evaluating the probability of *success* of a *K*oo*N* system is derived by conditioning on the state of the N-th component [3]. The N-th component can assume only two states, *success* with probability p_N and *failed* with probability q_N. Let us assume that the N-th component is correctly working: for a *K*oo*N* system to be correctly operating, we need at least $K-1$ working components out of the remaining $N-1$. If, on the other hand, the N-th component is failed, we need at least K working components out of the remaining $N-1$ in order to have a correctly operating *K*oo*N* system. The probability of *success* of a *K*oo*N* block can then be recursively computed as:

$$\begin{aligned} &p_{KooN} = q_N \cdot p_{Koo(N-1)} + p_N \cdot p_{(K-1)oo(N-1)} \\ &p_{0ooI} = 1 \\ &p_{JooI} = 0 \quad \forall J > I \end{aligned} \tag{7}$$

- **Bridge**. In order to compute the probability of *success* of a bridge block, we apply the same decompositional approach used in the second set of formulas to compute the probability of *success* of a *K*oo*N* block. Let us analyze the bridge block by conditioning the status of component E. If E is failed, the state of the block is equal to *success* if either A and B or C and D are correctly operating, i.e., if the parallel of two series A, B and C, D is satisfied. The probability of occurrence of this first event is equal to the probability of *failure* of E. On the other hand, if E is correctly operating, the state of the block is equal to *success* if at least one component between A and C is correctly operating and if at least one component between B and D is correctly operating, i.e., if the series of two parallel A, C and B, D is satisfied. The probability of occurrence of this second event is equal to the probability of *success* of E. The probability of *success* of a bridge block can then be computed through the formula:

$$\begin{aligned} p_{bridge} = {} & p_E \cdot (1 - q_A \cdot q_C) \cdot (1 - q_B \cdot q_D) + \\ & + q_E \cdot (1 - (1 - p_A \cdot p_B) \cdot (1 - p_C \cdot p_D)) \end{aligned} \tag{8}$$

One could argue that the formulas to compute probability of *success* of series and parallel blocks are specific cases of the *K*oo*N* block: series block can be treated as a *N*oo*N* block, while parallel block can be treated as a 1oo*N*. On the other hand, the mathematical representation for the specific cases of series and parallel blocks is simpler, thus justifying the usage of two additional formulas.

2.2.2. Quantitative Evaluation: Identical Components' Formulas

Under the assumption of N identical components having probability of *success* p, the following simplified formulas can be used to evaluate the probability of *success* of an RBD block:

- **Series**. By substituting p_i with p in Equation (2), we can compute the simplified probability of *success* of the series block p_{series} as:

$$p_{series} = p^N \tag{9}$$

- **Parallel**. By substituting p_i with p in Equation (4), we can compute the simplified probability of *success* of the parallel block $p_{parallel}$ as:

$$p_{parallel} = 1 - (1 - p)^N \tag{10}$$

- ***K*-out-of-*N* (*KooN*)**. By substituting p_i with p in Equations (5) and (6), we can compute the simplified probability of *success* of the *K*-out-of-*N* block R_{KooN} as:

$$p_{KooN} = \sum_{J=K}^{N} \binom{N}{J} \cdot p^J \cdot (1 - R)^{N-J} \tag{11}$$

- **Bridge**. By substituting p_A to p_E with p and q_A to q_E with q in (8), we obtain simplified probability of *success* of the bridge block p_{bridge} as:

$$p_{bridge} = p \cdot (1 - q^2)^2 + q \cdot (1 - (1 - p^2)^2) \tag{12}$$

2.3. Reliability Evaluation Using RBDs

The same mathematics described in Section 2.2 can be used to analytically compute the reliability curve of a block given the reliability curves of its components. In order to perform this time-dependent analysis, it is sufficient to replace each occurrence of p_x and q_x in equations from Equation (2) to Equation (12) with, respectively, $R_x(t)$ and $F_x(t)$.

The statement above is trivial and it is justified as follows: recall the probabilistic definition of reliability, i.e., the probability that the state of a given system or component at a given time t is equal to *success* given that it was correctly operating at the initial time t_0. We can then apply the same mathematics in Section 2.2 to quantitatively evaluate the probability that the state of an RBD block is equal to *success* at time t, i.e., its reliability.

The described approach can be easily adapted to those applications for which the analytical reliability curve is not needed but only samples acquired from it are sufficient. For example, the reliability curve of a system can be sampled at time instants $t_0 + k \cdot \Delta t$, where t_0 is the initial time, $k \in \mathbb{N}$ and Δt is the sampling period, by sampling the reliability curves of its components and by applying the proper equations for each evaluated time instant.

3. RBD Computation Library-librbd

As already stated in Section 1, our aim was to develop a library with the following characteristics:

- To be highly optimized;
- To support the most common OSes, i.e., Windows, Mac OS and Linux;
- To support the numerical computation of the reliability curve for all RBD basic blocks;
- To be available as a free software.

In order to meet the third goal, librbd implements the resolution formulas for series, parallel, *KooN* and bridge RBD blocks over time by accepting the following parameters:

- Number N of components within the block;
- Number T of temporal instants to be analyzed;
- Reliability values R for the modeled components over the requested time instants.

In order to meet the first two goals, several optimizations have been designed and implemented, as described in Section 3.1. Finally, in Section 3.2 we validate the results obtained using librbd by comparing the reliability curves of several blocks with the ones obtaining by using SHARPE tool.

3.1. Design

The two goals of portability and optimization tend to be in contrast. Interpreted languages, for example, are portable by nature, but they often lack performance; compiled languages, on the other hand, offer greater performance, but they are less portable when an interaction with the OS is required [46]. We decided to implement librbd in C language, at the cost of introducing small parts of conditional compilation when an interaction with the OS is needed. Furthermore, librbd is available both as a dynamic and static library.

In order to minimize numerical errors, all computations are performed using double-precision floating-point format (*double*) compliant with binary64 format [47].

Both the uncertainties and the numerical errors are due to the chosen format and, since the reliability is a real number in range $[0, 1]$, they are limited to the maximum resolution of floating-point numbers in the same range as described in [47].

We decided to implement both formulas for RBD blocks with identical components and for RBD blocks with generic components. This choice implies doubling the Application Programming Interfaces (APIs), thus almost doubling the size of the library itself, but it allows for the achievement of higher performance in the identical case, especially for *KooN* blocks.

3.1.1. Optimizations for *KooN* Computation

Several optimizations have been designed and implemented for RBD *KooN* blocks.

Two trivial optimizations, applicable to both RBD *KooN* blocks with generic and identical components, have been implemented. A *KooN* system with $K = N$ is solved as an RBD series block, while a *KooN* system with $K = 1$ is solved as an RBD parallel block.

A major optimization, applicable to both RBD *KooN* blocks with generic and identical components, minimizes the number of computational steps. This optimization exploits the formula of the trivial configuration 0oo$N$, which is shown in Equation (13):

$$R_{0ooN} = \sum_{i=0}^{N} \sum_{j=1}^{\binom{N}{i}} P_{path(N,i,j)} = 1 \tag{13}$$

Starting from Equation (13), we divide the outer sum into two separate sums, the first one ranging from 0 to $K-1$, the second one ranging from K to N, and we substitute the contribution shown in Equation (6). Finally, we resolve for R_{KooN} as:

$$\begin{aligned} \sum_{i=0}^{K-1} \sum_{j=1}^{\binom{N}{i}} P_{path(N,i,j)} + \sum_{i=K}^{N} \sum_{j=1}^{\binom{N}{i}} P_{path(N,i,j)} &= 1 \\ F_{KooN} + R_{KooN} &= 1 \\ R_{KooN} &= 1 - F_{KooN} \end{aligned} \tag{14}$$

where F_{KooN} is the unreliability of a *KooN* block, i.e., the probability of having at least $N - K + 1$ components *failed* in a block of N components. We finally observe that, when $N - K > K - 1$, we can compute R_{KooN} exploiting Equation (14) decreasing the mathematical complexity.

For RBD *KooN* blocks with identical components, we compute and store all coefficients $\binom{N}{i}$, $\forall i \in [K, N]$ that will be used during the computation of the reliability for each time instant. For RBD *KooN* blocks with generic components, we try to compute all combinations of i out of N components with $i \in [K, N]$ that are needed to compute the reliability for each time instant. The number of these combinations is equal to $\sum_{i=K}^{N} \binom{N}{i}$.

The last optimization, which is applicable to RBD *KooN* blocks with generic components, is the adoption of a heuristic to decrease the computation time by using either Equation (7) or Equation (14). The number of recursion steps performed while applying Equation (7) is limited to N^2 [41]. On the other hand, the number of iterative steps performed while applying Equation (14) is limited to $\sum_{i=K}^{N} \binom{N}{i}$, with $K \geq N/2$. The chosen heuristic used to compute reliability of a *KooN* block with generic components is the following one:

- Use Equation (7) when all the following conditions are true:
  - The OS is able to allocate the memory to store all combinations of i out of N components with $i \in [K, N]$ that are needed to compute the reliability for each time instant;
  - $\sum_{i=K}^{N} \binom{N}{i} < N^2$.
- Use Equation (14) otherwise.

Algorithm 1, together with auxiliary functions shown in Algorithm 2, is used to compute the reliability of an RBD *KooN* block with generic components, while Algorithm 3 is used to compute the reliability of an RBD *KooN* block with identical components.

Algorithm 1: Computation of RBD *KooN* block with generic components.

```
Input: Reliability R_i of each component
Result: Reliability R of KooN block
begin
    N_square = N · N;
    sum_nCi = 0;
    if (N − K) ≤ (K − 1) then
        for i ∈ [K, N] do
            sum_nCi = sum_nCi + (N i);
        if sum_nCi <= N_square then
            R = 0;
            for i ∈ [K, N] do
                for j ∈ (N i) do
                    R = R + ReliabilityStep(N, i, j);
        else
            R = ReliabilityRecursive(N, K);
    else
        for i ∈ [0, K − 1] do
            sum_nCi = sum_nCi + (N i);
        if sum_nCi <= N_square then
            R = 1;
            for i ∈ [0, (K − 1)] do
                for j ∈ (N i) do
                    R = R − ReliabilityStep(N, i, j);
        else
            R = ReliabilityRecursive(N, K);
```

Algorithm 2: Auxiliary functions for computation of RBD *KooN* block with generic components.

Input: Reliability R_i of each component
Input: j-th combination of $i$oo$N$ components $C(N, i, j)$
Function *ReliabilityStep(N, i, j)*
  $R_{step} = 1$;
  for $l \in [1, N]$) **do**
    if $l \in C(N, i, j)$ **then**
      $R_{step} = R_{step} \cdot R_l$;
    else
      $R_{step} = R_{step} \cdot (1 - R_l)$;
  return R_{step};
Function *ReliabilityRecursive(i, j)*
  if $j = 0$ **then**
    return 1;
  if $j > i$ **then**
    return 0;
  return $(1 - R_i) \cdot ReliabilityRecursive(i - 1, j) +$
  $R_i \cdot ReliabilityRecursive(i - 1, j - 1)$;

Algorithm 3: Computation of RBD *KooN* block with identical components.

Input: Reliability R_c of each component
Result: Reliability R of *KooN* block
begin
  if $(N - K) \leq (K - 1)$ **then**
    $R = 0$;
    for $i \in [K, N]$ **do**
      $R = R + \binom{N}{i} \cdot R_c^i \cdot (1 - R_c)^{N-i}$;
  else
    $R = 1$;
    for $i \in [0, (K - 1)]$ **do**
      $R = R - \binom{N}{i} \cdot R_c^i \cdot (1 - R_c)^{N-i}$;

3.1.2. Symmetric Multi-Processing (SMP)

In order to further increase performance, librbd adopts the Symmetric Multi-Processing (SMP) paradigm. The external library chosen for adding SMP support is *pthreads*. This library implements the management of threads and is compliant with the POSIX standard OS interface [48]. This library is always available on fully and mostly POSIX-compliant OSes (e.g., Mac OS and Linux). Microsoft Windows does not offer a native support to pthreads, but it is still possible to use it through one of the following two methods:

- Download pthreads-win32, a freely available library which implements a large subset of the POSIX standard threads related API for Windows [49]. After pthreads-win32 has been downloaded, it is possible to use the desired IDE and Compiler (e.g., Visual Studio).
- Download and install Cygwin, a freely available environment (i.e., tools and libraries) which provides a large collection of GNU and Open Source tools, including GCC, and a substantial POSIX API functionality, including pthreads-win32 [50].

In order to fully exploit the SMP paradigm, a key point is the subdivision of the task into batches. For this particular problem, the best subdivision is to assign a subset of data, i.e., a batch, to each thread. In particular, each thread receives as input the reliability values of all components over a subset of time instants. Furthermore, librbd interrogates the OS to retrieve the total number of CPU cores and uses this number as the maximum number of threads that can be created.

The usage of the SMP paradigm adds an overhead: each time an application requests the creation of a new thread and each time a thread terminates its computation, the OS has to perform additional operations. This overhead negates the benefits of SMP when the computational task is too small. In order to mitigate this issue, several tests have been conducted to find a minimum to the batch size. This minimum has been empirically set to 10,000 time instants.

The SMP functionality can be enabled or disabled at compile time. When SMP is not needed, i.e., when librbd is built as a Single Threaded (ST) library, it is compiled providing external compiler flag `CPU_SMP` defined with value 0. When SMP is needed, librbd is compiled without providing external compiler flag `CPU_SMP` or by defining it with a value different from 0.

3.2. Validation

To validate the developed library, we perform a comparison of librbd output with the one obtained by using SHARPE tool [41]. Two different RBD models for each RBD block have been generated, one using identical components and the other using generic components. The validation process has been carried out by using fifteen electronics components with constant failure rate validated using Telcordia SR-332 [51]: their failure rate is shown in Table 1. The eight RBD blocks modeled during the validation process are shown in Table 2.

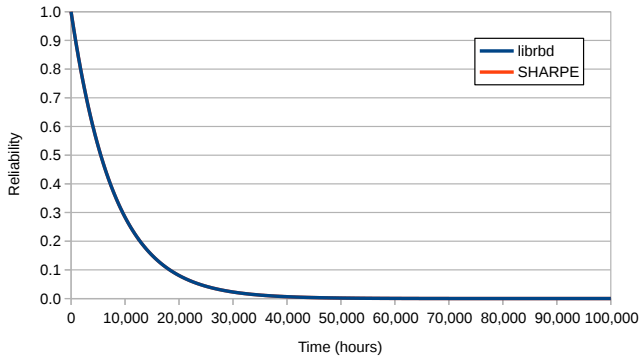
Table 1. Components and their respective failure rate λ.

Component	Failure Rate λ (h^{-1})	Component	Failure Rate λ (h^{-1})
C1	0.0000084019	C9	0.0000027777
C2	0.0000039438	C10	0.0000055397
C3	0.0000078310	C11	0.0000047740
C4	0.0000079844	C12	0.0000062887
C5	0.0000091165	C13	0.0000036478
C6	0.0000019755	C14	0.0000051340
C7	0.0000033522	C15	0.0000095223
C8	0.0000076823		

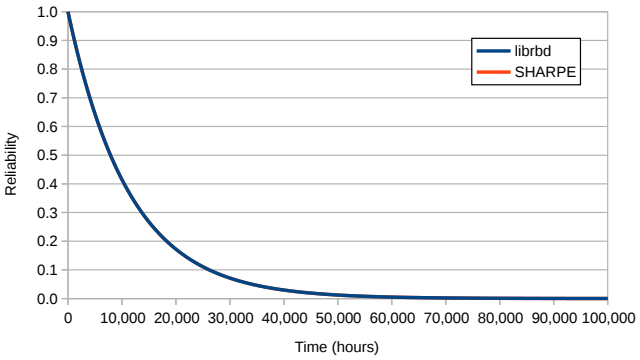
Table 2. RBD models used during validation.

RBD Block	Topology	T (h)	Components
Series identical	15 components	200,000	C1
Series generic	15 components	200,000	All
Parallel identical	15 components	200,000	C1
Parallel generic	15 components	200,000	All
KooN identical	8oo15	200,000	C1
KooN generic	8oo15	200,000	All
Bridge identical	5 components	200,000	C1
Bridge generic	5 components	200,000	From C1 to C5

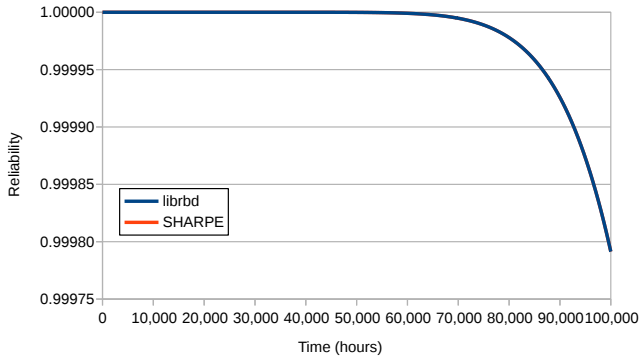
For each RBD model, we have produced an output file containing the reliability of each analyzed time instant using both librbd and SHARPE. The reliability has been formatted using scientific notation with eight decimal places. Please note that, due to the chosen numerical representation, the uncertainty of the comparison operations is limited to 1.00×10^{-8}. The reliability curves obtained for each analyzed block are shown in Figure 4a–h. For a better graphical visualization, only the first 100,000 h are plotted for each reliability curve. The blue line represents the reliability curve obtained using librbd, while the red one uses SHARPE tool. In all the figures, the blue and red curves are overlapping, with the blue line completely hiding the red one. The visual inspection of the reliability curves obtained from the RBD models with both librbd and SHARPE tool proves the validity of the developed library.



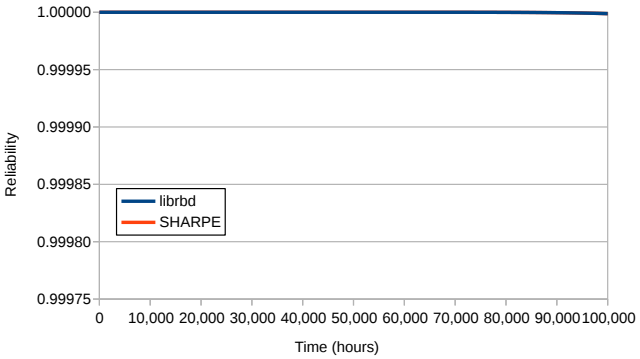
(**a**) Series identical.



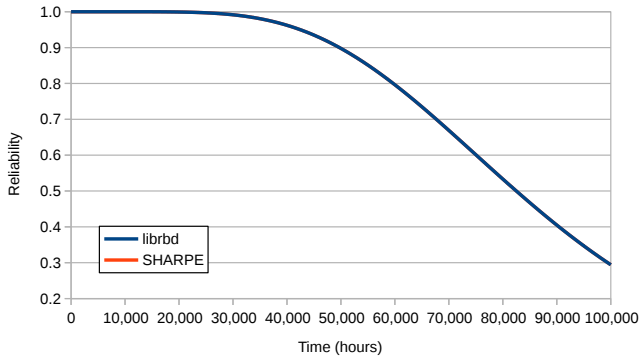
(**b**) Series generic.



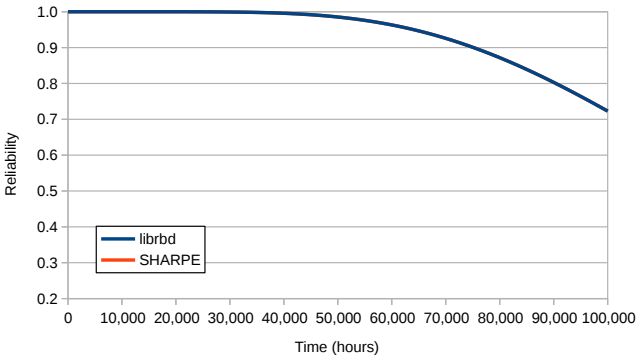
(**c**) Parallel identical.



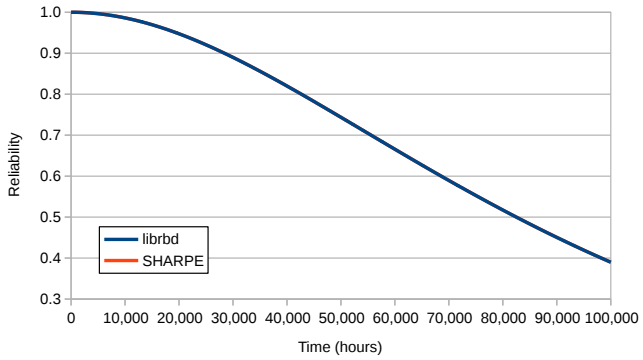
(**d**) Parallel generic.



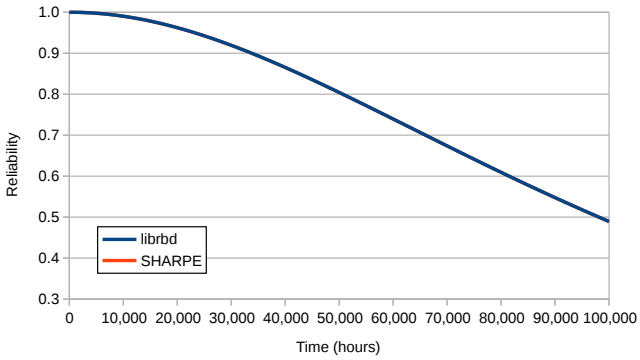
(**e**) Koo*N* identical.



(**f**) Koo*N* generic.



(**g**) Bridge identical.



(**h**) Bridge generic.

Figure 4. Validation of reliability of RBD blocks.

To further strengthen the results of this validation we have computed, for each analyzed RDB model, the error function as the difference between the two reliability curves and we have computed the maximum and minimum value of this error function. The obtained minimum and maximum error are shown, for each RBD model, in Table 3. We suppose that this error is due to the different implementation of the two tools, in particular regarding the exact sequence of operations performed over floating-point numbers. From the analysis of the obtained results, we can observe that the maximum and minimum error between the two solutions is lower than the maximum uncertainty, hence it can be considered negligible.

Hence, we can conclude that librbd produces the valid result, i.e., the correct reliability curve, for each implemented RBD basic block.

Table 3. Error function between librbd and SHARPE tool.

RBD Block	Error Function	
	Minimum	**Maximum**
Series identical	-1.00×10^{-13}	1.00×10^{-13}
Series generic	-1.00×10^{-13}	1.00×10^{-11}
Parallel identical	0.00×10^{0}	0.00×10^{0}
Parallel generic	0.00×10^{0}	0.00×10^{0}
KooN identical	-1.00×10^{-9}	0.00×10^{0}
KooN generic	-1.00×10^{-9}	0.00×10^{0}
Bridge identical	0.00×10^{0}	0.00×10^{0}
Bridge generic	0.00×10^{0}	0.00×10^{0}

3.3. librbd Usage

This section covers the library API. As already stated, *librbd* provides, for each RBD basic block, two distinct interfaces, one for the case of generic components and the other one for identical components. The library exposes a single header file, "rbd.h", which provides access to the entire API.

3.3.1. API for Generic Components

The following interfaces are used to evaluate the reliability of RBD blocks with generic components:

- int *rbdSeriesGeneric*(double ***R**, double ***O**, unsigned char **N**, unsigned int **T**)
- int *rbdParallelGeneric*(double ***R**, double ***O**, unsigned char **N**, unsigned int **T**)
- int *rbdKooNGeneric*(double ***R**, double ***O**, unsigned char **N**, unsigned char **K**, unsigned int **T**)
- int *rbdBridgeGeneric*(double ***R**, double ***O**, unsigned char **N**, unsigned int **T**)

The capitalized and bold characters identify the following input parameters:

- **N**: the number of components inside the block. Note that, for the bridge block, the number of components must be equal to 5.
- **K**: the minimum number of components inside the block. Note that this parameter is available for *KooN* blocks.
- **T**: the number of time instants over which the reliability curve is computed.
- **R**: the reliability curves for the input components. Since this is the generic components case, this parameter underlies an $\mathbf{N} \times \mathbf{T}$ matrix.
- **O**: the computed reliability curve, returned as an array of **T** elements.

All interfaces return 0 in case of successful computation and a negative number otherwise.

3.3.2. API for Identical Components

The following interfaces are used to evaluate the reliability of RBD blocks with identical components:

- int *rbdSeriesIdentical*(double ***R**, double ***O**, unsigned char **N**, unsigned int **T**)
- int *rbdParallelIdentical*(double ***R**, double ***O**, unsigned char **N**, unsigned int **T**)

- int *rbdKooNIdentical*(double *__R__, double *__O__, unsigned char __N__, unsigned char __K__, unsigned int __T__)
- int *rbdBridgeIdentical*(double *__R__, double *__O__, unsigned char __N__, unsigned int __T__)

The capitalized and bold characters identify the following input parameters:

- **N**: the number of components inside the block. Note that, for the bridge block, the number of components must be equal to 5.
- **K**: the minimum number of components inside the block. Note that this parameter is available for *KooN* blocks.
- **T**: the number of time instants over which the reliability curve is computed.
- **R**: the reliability curves for the input components. Since this is the identical components case, this parameter underlies an array of **T** elements.
- **O**: the computed reliability curve, returned as an array of **T** elements.

All interfaces return 0 in case of successful computation, a negative number otherwise.

3.3.3. Example of librbd Usage

Let us consider the system shown in Figure 1. Since components C1 and C2 are identical in parallel configuration, the reliability curve of this first block can be computed by invoking `rbdParallelIdentical`. Finally, the reliability of the whole system can be computed through a series block with components C3 and C4 and the reliability of the previously evaluated parallel block by invoking `rbdSeriesGeneric`.

Listing 1 shows the C source code used to implement and analyze the described system. The reliability curves of all components are evaluated over $T = 10$ time instants. The source code, through the consecutive invocation of two different librbd APIs, evaluates the reliability of the whole system. Finally, the evaluated reliability is printed to the standard output.

The C source code related to the described example and shown in Listing 1 has been compiled and executed. Its output is shown in Listing 2.

4. Materials and Methods

In this section, we present the materials and the methods used to evaluate the performance of librbd. In particular, in Section 4.1 we present the materials used, while in Section 4.2 we discuss the methodology adopted to evaluate the performance of librbd and to compare it with SHARPE.

4.1. Materials

The five PCs listed in Table 4 have been used in order to measure the performance of librbd [43].

Both librbd and test binaries were built using the following compiling options:

- Optimization level set to the maximum level ($-$O3);
- Target architecture set as follows:
  - Advanced Micro Devices X86-64 (amd64) for Intel CPUs;
  - ARMv6 with VFPv2 coprocessor (armv6+fp) for ARM CPU.

To compare the performance of librbd, we use SHARPE [41,42] since it is the closest tool to all our requirements. SHARPE tool is available on Windows OS only. To perform the comparison with SHARPE tool, we exploited the dual boot-able PC1. The second OS installed on this machine is Windows 7 with Cygwin GCC 7.4.0.

Listing 1: Example of librbd usage.

```
#include <stdio.h>
#include <string.h>
#include "rbd.h"     /* Include librbd header                          */

void main (void){
   /* Input data r_c1 - Reliability of C1/C2 as T array               */
   double r_c1[10] = {
      1.000, 0.930, 0.860, 0.790, 0.720,
      0.650, 0.580, 0.510, 0.440, 0.370
   };
   /* Input data r_c3 - Reliability of C3 as T array                  */
   double r_c3[10] = {
      1.000, 0.980, 0.960, 0.940, 0.920,
      0.900, 0.880, 0.860, 0.840, 0.820
   };
   /* Input data r_c4 - Reliability of C4 as T array                  */
   double r_c4[10] = {
      1.000, 0.970, 0.950, 0.910, 0.880,
      0.860, 0.830, 0.780, 0.720, 0.610
   };
   /* Intermediate data r_tmp - Reliability as NxT matrix             */
   double r_tmp[3][10];
   /* Output data - Reliability of the system as T array              */
   double r_system[10];

   /* Compute reliability of parallel block and store                 */
   /* result in first row of r_tmp                                    */
   rbdParallelIdentical (&r_c1[0], &r_tmp[0][0], 2, 10);

   /* Copy reliability of C3 to second row of r_tmp                   */
   memcpy (&r_tmp[1][0], &r_c3[0], sizeof(double) * 10);
   /* Copy reliability of C4 to third row of r_tmp                    */
   memcpy (&r_tmp[2][0], &r_c4[0], sizeof(double) * 10);

   /* Compute reliability of series block and store                   */
   /* result in r_system                                              */
   rbdSeriesGeneric (&r_tmp[0][0], &r_system[0], 3, 10);

   /* Print computed reliability                                      */
   for(int i = 0; i < 10; i++){
      printf ("Reliability %d:  %.6f\n", i, r_system[i]);
   }
}
```

Listing 2: Output of librbd example.

```
Reliability 0:  1.000000
Reliability 1:  0.945942
Reliability 2:  0.894125
Reliability 3:  0.817677
Reliability 4:  0.746127
Reliability 5:  0.679185
Reliability 6:  0.601557
Reliability 7:  0.509741
Reliability 8:  0.415135
Reliability 9:  0.301671
```

Table 4. PCs used for performance evaluation.

Name	Chassis	CPU & RAM	OS & Compiler
PC1	Workstation	Intel i7-2600 @ 3.8GHz 16GB-DDR3 @ 1333MHz	Ubuntu 18.04_amd64 GCC 5.4.0
PC2	Notebook	Intel i7-6700HQ @ 3.5GHz 16GB-LPDDR3 @ 2133MHz	Mac OS 10.13.6 Apple LLVM 10.0.0
PC3	Notebook	Intel i7-7700HQ @ 3.8GHz 32GB-DDR4 @ 2400MHz	Ubuntu 18.04_amd64 GCC 5.4.0
PC4	Notebook	Intel i5-8365U @ 1.9GHz 16GB-DDR4 @ 2666MHz	Windows 10 Cygwin GCC 7.4.0
PC5	Raspberry Pi 3	4 × Cortex-A53 @ 1.2GHz 1GB-LPDDR2 @ 900MHz	Raspberry Pi OS 10_AArch32 GCC 8.3.0

4.2. Methods

The evaluation of performance has been conducted through the usage of a test application instrumented in order to measure the execution time of librbd of several RDB blocks over different time instant configurations. In particular, we defined the following set of RBD models to be used during the execution time monitoring:

- Series blocks with 2, 3, 5, 10 and 15 generic components;
- Series blocks with 2, 3, 5, 10 and 15 identical components;
- Parallel blocks with 2, 3, 5, 10 and 15 generic components;
- Parallel blocks with 2, 3, 5, 10 and 15 identical components;
- 1oo2, 2oo3, 3oo5, 5oo10 and 8oo15 blocks with generic components;
- 1oo2, 2oo3, 3oo5, 5oo10 and 8oo15 blocks with identical components;
- Bridge block with generic components;
- Bridge block with identical components.

Each RBD model has been analyzed for 50,000, 100,000 and 200,000 time instants. To further investigate the performance of *KooN* RBD blocks, we used the following models and we analyzed them over 100,000 time instants:

- All blocks ranging from 1oo15 to 15oo15 with generic components;
- All blocks ranging from 1oo15 to 15oo15 with identical components.

The chosen instrumentation method has been implemented through a succession of invocations to `clock_gettime` API, one immediately preceding the librbd invocation and the other one as soon as librbd returned. The usage of this API has the following advantages and drawbacks:

- It returns a monotonic clock (i.e., guaranteed to be nondecreasing), which allows to measure the time spent in executing a program routine with a time resolution up to nanoseconds;
- It takes into account not only user time (application time) but also system time;
- It is defined by The Open Group POSIX standard [48].

In order to minimize the impact from the second point, each experiment has been repeated 15 times and, after all experiments were run, the median time of execution has been selected. We chose the median time since it minimizes the impact of the time spent by the OS which is unrelated to the RBD block analysis.

To compare the performance of librbd w.r.t. SHARPE, we define the following set of RBD models analyzed for 100,000 time instants:

- Series blocks with 2, 3, 5, 10 and 15 generic components;
- Series blocks with 2, 3, 5, 10 and 15 identical components;
- Parallel blocks with 2, 3, 5, 10 and 15 generic components;
- Parallel blocks with 2, 3, 5, 10 and 15 identical components;
- 1oo2, 2oo3, 3oo5, 5oo10 and 8oo15 blocks with generic components;
- 1oo2, 2oo3, 3oo5, 5oo10 and 8oo15 blocks with identical components;
- Bridge block with generic components;
- Bridge block with identical components.

The comparison experiments were done for series, parallel, *K*oo*N* and bridge blocks for a time interval of 200,000 time instants. The failure rate λ of each component has been chosen using the same criteria described in Section 3.2 and in Table 1.

To compare the execution time, we created a test application that uses librbd to run the aforementioned RBD models and that produces an output log file with the same formatting of SHARPE tool.

5. Results and Discussion: Evaluation of librbd Performance

In this section, we evaluate the performance of the librbd library. In Section 5.1 we evaluate the execution time of librbd using different computers and different RBD layouts. Finally, in Section 5.2, we compare the execution time needed to analyze the same problem by both SHARPE and librbd.

5.1. Evaluation of librbd Execution Time

The measurement experiments were performed as described in Section 4.2 and the execution time results are presented in Tables 5–14 for generic and identical components, for each of the different RBD blocks (namely, series, parallel, *N*/2oo*N*, *K*oo15, bridge).

As expected, for both series and parallel blocks with generic and identical components (see Tables 5–8), we observe that the computation time is linear w.r.t. both the number of time intervals and the number of components.

This result hence confirms the correctness of librbd design w.r.t. the optimization of both series and parallel blocks.

Table 5. Performance evaluation of series generic block.

Topology	# Times	Execution Time (ms)				
		PC1	PC2	PC3	PC4	PC5
2	50,000	0.436	0.268	0.228	0.657	2.812
2	100,000	0.247	0.365	0.290	0.896	4.336
2	200,000	0.548	0.706	0.621	1.692	8.156
3	50,000	0.174	0.260	0.228	0.671	2.446
3	100,000	0.380	0.419	0.307	0.904	4.732
3	200,000	0.701	0.851	0.657	1.659	11.720
5	50,000	0.193	0.264	0.289	0.646	4.229
5	100,000	0.412	0.394	0.436	0.927	6.505
5	200,000	0.902	0.792	0.729	1.767	13.761
10	50,000	0.351	0.351	0.317	0.706	8.312
10	100,000	0.884	0.637	0.634	1.021	13.734
10	200,000	1.754	1.190	1.144	1.972	27.330
15	50,000	0.538	0.491	0.433	0.760	11.697
15	100,000	1.279	0.873	0.848	1.117	20.715
15	200,000	2.650	1.831	1.646	2.034	41.191

Table 6. Performance evaluation of series identical block.

Topology	# Times	Execution Time (ms)				
		PC1	PC2	PC3	PC4	PC5
2	50,000	0.388	0.263	0.220	0.633	2.325
2	100,000	0.252	0.343	0.290	0.958	3.785
2	200,000	0.511	0.638	0.597	1.709	6.790
3	50,000	0.163	0.255	0.205	0.690	1.510
3	100,000	0.365	0.370	0.306	0.869	3.521
3	200,000	0.602	0.774	0.626	1.558	6.591
5	50,000	0.179	0.260	0.253	0.683	1.865
5	100,000	0.329	0.356	0.411	0.856	3.098
5	200,000	0.561	0.690	0.694	1.740	6.862
10	50,000	0.287	0.267	0.274	0.667	2.146
10	100,000	0.532	0.379	0.438	0.925	4.041
10	200,000	0.673	0.725	0.737	1.839	8.507
15	50,000	0.413	0.314	0.347	0.737	2.419
15	100,000	0.573	0.469	0.565	0.932	4.262
15	200,000	0.874	0.933	0.935	1.874	7.831

Table 7. Performance evaluation of parallel generic block.

Topology	# Times	Execution Time (ms)				
		PC1	PC2	PC3	PC4	PC5
2	50,000	0.197	0.310	0.204	0.685	1.978
2	100,000	0.278	0.430	0.392	0.858	4.053
2	200,000	0.594	0.861	0.635	1.588	8.407
3	50,000	0.183	0.295	0.198	0.655	3.190
3	100,000	0.356	0.432	0.456	0.884	5.626
3	200,000	0.691	0.804	0.663	1.698	11.161
5	50,000	0.206	0.313	0.226	0.686	4.056
5	100,000	0.474	0.433	0.456	0.951	7.139
5	200,000	0.940	0.858	0.734	1.655	14.948
10	50,000	0.446	0.355	0.419	0.651	9.915
10	100,000	0.879	0.617	0.709	0.947	16.667
10	200,000	1.797	1.232	1.110	1.891	29.725
15	50,000	0.548	0.440	0.552	0.765	13.544
15	100,000	1.347	0.945	0.896	1.089	22.388
15	200,000	2.649	1.853	1.916	2.110	42.094

Table 8. Performance evaluation of parallel identical block.

Topology	# Times	Execution Time (ms)				
		PC1	PC2	PC3	PC4	PC5
2	50,000	0.181	0.302	0.199	0.641	1.406
2	100,000	0.260	0.396	0.342	0.838	3.983
2	200,000	0.523	0.821	0.629	1.627	7.381
3	50,000	0.178	0.292	0.197	0.642	1.637
3	100,000	0.360	0.423	0.444	0.831	3.316
3	200,000	0.566	0.756	0.623	1.630	6.872
5	50,000	0.187	0.292	0.224	0.704	2.093
5	100,000	0.401	0.383	0.438	0.868	3.784
5	200,000	0.594	0.776	0.673	1.527	7.597
10	50,000	0.290	0.284	0.315	0.627	2.068
10	100,000	0.561	0.394	0.489	0.854	4.145
10	200,000	0.718	0.754	0.718	1.700	7.429
15	50,000	0.388	0.305	0.347	0.688	2.522
15	100,000	0.642	0.476	0.552	0.918	4.626
15	200,000	0.801	0.892	0.951	1.997	8.311

Table 9. Performance evaluation of $N/2ooN$ generic block.

Topology	# Times	Execution Time (ms)				
		PC1	PC2	PC3	PC4	PC5
1oo2	50,000	0.183	0.306	0.211	0.667	1.810
1oo2	100,000	0.254	0.402	0.393	0.791	3.873
1oo2	200,000	0.561	0.782	0.617	1.540	8.123
2oo3	50,000	0.669	0.768	0.709	0.776	6.478
2oo3	100,000	1.265	1.278	1.280	1.163	12.146
2oo3	200,000	2.724	2.299	2.787	2.122	21.686
3oo5	50,000	2.453	2.419	2.716	2.103	30.528
3oo5	100,000	4.313	5.402	4.468	4.090	44.617
3oo5	200,000	7.224	9.466	8.448	7.611	82.247
5oo10	50,000	20.584	45.096	20.462	35.188	299.916
5oo10	100,000	44.083	88.876	46.227	78.124	506.973
5oo10	200,000	78.657	178.793	80.726	288.728	986.001
8oo15	50,000	558.292	1212.422	571.278	1838.166	7753.987
8oo15	100,000	1167.856	2406.377	1194.093	3694.708	15,316.800
8oo15	200,000	2233.662	4784.158	2278.303	7666.604	30,172.290

Table 10. Performance evaluation of $N/2ooN$ identical block.

Topology	# Times	Execution Time (ms)				
		PC1	PC2	PC3	PC4	PC5
1oo2	50,000	0.171	0.293	0.199	0.617	1.397
1oo2	100,000	0.250	0.378	0.367	0.805	3.613
1oo2	200,000	0.533	0.748	0.598	1.537	8.085
2oo3	50,000	0.377	0.334	0.380	0.649	3.745
2oo3	100,000	0.616	0.482	0.718	1.002	6.674
2oo3	200,000	0.980	0.856	1.149	2.099	12.279
3oo5	50,000	0.491	0.372	0.579	0.964	5.452
3oo5	100,000	0.840	0.682	0.882	1.148	9.192
3oo5	200,000	1.251	1.244	1.513	2.056	15.876
5oo10	50,000	0.658	0.708	0.764	0.961	10.244
5oo10	100,000	1.315	1.396	1.220	1.309	15.746
5oo10	200,000	2.169	2.339	1.990	4.621	28.266
8oo15	50,000	1.422	1.168	1.720	2.651	18.303
8oo15	100,000	3.027	2.456	2.810	4.756	27.901
8oo15	200,000	4.403	4.035	4.940	10.613	48.033

Table 11. Performance evaluation of $Koo15$ generic block.

Topology	# Times	Execution Time (ms)				
		PC1	PC2	PC3	PC4	PC5
1oo15	100,000	1.280	0.765	0.929	2.305	17.021
2oo15	100,000	10.944	10.057	9.167	24.552	88.657
3oo15	100,000	73.165	61.577	64.344	155.191	571.809
4oo15	100,000	183.721	341.797	183.270	604.827	2025.680
5oo15	100,000	406.931	836.953	418.733	1297.444	4983.553
6oo15	100,000	740.532	1503.090	753.573	2314.167	9237.214
7oo15	100,000	1062.330	2126.853	1071.369	3260.557	13,325.713
8oo15	100,000	1190.309	2386.732	1192.385	4171.043	15,280.717
9oo15	100,000	1064.583	2134.690	1052.093	3688.210	13,398.969
10oo15	100,000	742.056	1511.135	758.824	2466.044	9164.543
11oo15	100,000	411.900	837.570	417.853	1287.806	4967.779
12oo15	100,000	183.203	341.557	189.317	590.764	2015.668
13oo15	100,000	62.120	107.026	72.951	107.102	664.601
14oo15	100,000	9.432	15.416	10.368	17.220	100.526
15oo15	100,000	1.440	0.933	1.041	2.163	18.609

Table 12. Performance evaluation of *K*oo15 identical block.

Topology	# Times	Execution Time (ms)				
		PC1	PC2	PC3	PC4	PC5
1oo15	100,000	0.663	0.430	0.635	2.083	4.653
2oo15	100,000	0.828	0.883	0.884	2.908	8.284
3oo15	100,000	1.154	1.185	1.064	3.054	11.596
4oo15	100,000	1.454	1.401	1.402	3.385	14.935
5oo15	100,000	1.848	1.660	1.535	3.183	18.000
6oo15	100,000	2.147	1.898	1.723	3.513	21.169
7oo15	100,000	2.478	2.136	1.968	3.734	24.497
8oo15	100,000	3.063	2.475	2.831	5.842	27.787
9oo15	100,000	2.731	2.118	2.596	6.283	24.446
10oo15	100,000	2.432	1.900	2.418	4.917	21.156
11oo15	100,000	2.072	1.646	1.931	3.696	18.043
12oo15	100,000	1.562	1.414	1.666	3.144	14.822
13oo15	100,000	1.240	1.181	1.340	2.564	11.546
14oo15	100,000	0.933	0.935	1.128	2.342	8.221
15oo15	100,000	0.635	0.464	0.641	1.980	4.719

Table 13. Performance evaluation of bridge generic block.

# Times	Execution Time (ms)				
	PC1	PC2	PC3	PC4	PC5
50,000	0.253	0.263	0.240	1.361	2.962
100,000	0.406	0.449	0.452	1.736	5.967
200,000	0.853	0.823	0.774	3.375	10.985

Table 14. Performance evaluation of bridge identical block.

# Times	Execution Time (ms)				
	PC1	PC2	PC3	PC4	PC5
50,000	0.235	0.248	0.218	1.377	1.600
100,000	0.338	0.341	0.395	1.673	3.272
200,000	0.548	0.706	0.649	3.293	7.289

The experiments with $N/2$oo$N$ blocks are used to evaluate the worst case execution time of *K*oo*N* blocks (see Tables 9–10). Using the optimized computation Algorithms 1 and 2 for *K*oo*N* blocks with generic components we verify that the complexity is $O(N^2 \cdot T)$ and that the complexity for *K*oo*N* blocks with identical components computed using Algorithm 3 is $O(N \cdot T)$. The experiments with *K*oo15 blocks are used to evaluate the impact of the optimized algorithms for *K*oo*N* computation with a fixed number of components (see Tables 11–12). For these experiments, we observe that the execution time is, as expected, symmetrical w.r.t. the worst case $K = N/2$.

The experiments performed on all *K*oo*N* blocks are aligned with the foreseen complexity and hence they validate the correctness of both librbd design and optimization algorithms implemented for the most complex RBD block.

Finally, as expected, for bridge block with both generic and identical components, we observe that the computation time is linear w.r.t. the number of time intervals (see Tables 13 and 14).

In addition, for this last result, we confirm the correctness of librbd design w.r.t. the optimization of bridge blocks.

Finally, it is important to note the execution times on the different PCs. We observe that, for all Intel i7 CPUs, we have obtained comparable results. The difference between the Intel i5 CPU and the other Intel CPUs may be due to both differences in the CPU architecture and in the OS used. Finally, we observe a sensible difference between all Intel-based PCs and the ARM-based one. This may be due to several architectural differences of the two CPU architectures. The principal one could be that, while the tested ARM CPU incorporates 4 physical cores, i.e., with the number of concurrently executing threads equal to 4, the tested Intel CPUs incorporate 4 physical cores with Hyper-Threading technology, thus

providing a number of concurrently executing threads ranging from 4 to 8. Nonetheless, the performed experiments show promising results also on low-power and low-cost CPUs.

5.2. Comparison with SHARPE

SHARPE provides a Graphical User Interface for specifying performance, reliability and performability models through the usage of a specification language. It then provides analysis and solution methods for the previously generated models. The analysis methods are available as a Command Line Interface executable. This tool acquires the input model through the usage of a SHARPE specification language file and produces as output a textual log file.

The comparison experiments were performed as described in Section 4.2 and the obtained results are presented in Table 15. We observe that the execution times of librbd are significantly lower than the ones obtained by using SHARPE.

Table 15. Execution time comparison.

RBD Block	Topology	Execution Time (s)		Gain (%)
		librbd	SHARPE	
Series generic	2	0.10	0.98	89.95
	3	0.10	1.12	91.04
	5	0.11	0.98	88.97
	10	0.10	1.94	94.60
	15	0.10	2.59	96.19
Series identical	2	0.10	0.90	89.24
	3	0.11	0.96	88.32
	5	0.11	0.90	87.72
	10	0.12	1.23	90.40
	15	0.10	1.45	93.31
Parallel generic	2	0.11	0.98	89.08
	3	0.10	1.12	91.20
	5	0.11	0.98	89.22
	10	0.10	2.03	94.95
	15	0.10	2.78	96.36
Parallel identical	2	0.10	0.92	89.19
	3	0.11	0.95	88.74
	5	0.11	0.92	88.51
	10	0.12	1.31	91.16
	15	0.10	1.59	93.85
KooN generic	1oo2	0.11	1.01	89.37
	2oo3	0.12	1.15	89.95
	3oo5	0.11	1.01	89.44
	5oo10	0.15	2.28	93.21
	8oo15	2.35	3.36	29.96
KooN identical	1oo2	0.11	0.88	87.80
	2oo3	0.11	0.89	88.13
	3oo5	0.11	0.88	87.71
	5oo10	0.08	1.31	94.20
	8oo15	0.08	1.81	95.71
Bridge generic	5	0.10	1.64	94.16
Bridge identical	5	0.09	1.66	94.35

6. Conclusions and Future Work

In this paper, we presented librbd, an open source optimized library for reliability evaluation using the RBD formalism. After the mathematical background description, we illustrated the library design and characteristics. Then, we showed its execution times on different platforms and performed a comparison with the most similar tool identified at the state-of-the-art. The good results achieved enable the usage of librbd in hierarchical approaches for reliability evaluation exploiting strengths of both combinatorial and state-space based models. In particular, the comparison experiments show that the execution time of librbd is, in general, almost 9 times faster than the one obtained using SHARPE.

These good results encourage the usage of librbd to implement a prognostics application that leverages a frequent computation of the tuned reliability curve.

Future research and advancements of librbd include the following:

- Development of a tool that allows the graphical definition of RBDs and their analysis through the usage of librbd.
- Development of a tool that allows the graphical definition of hierarchical models (RBDs and STPNs/GSPNs) and that integrates librbd and SIRIO library for their quantitative evaluation.
- Usage of native Windows APIs for SMP to decrease the execution time on this OS.
- Investigation on the performance loss experienced for *KooN* block with generic components. This performance loss is visible with $N = 15$ and it is probably due to a nonoptimized usage of the cache memory.

Author Contributions: Conceptualization, L.C. (Laura Carnevali), L.C. (Lorenzo Ciani), A.F. and M.P.; methodology, L.C. (Laura Carnevali), L.C. (Lorenzo Ciani), A.F. and M.P.; software, M.P.; validation, A.F., G.G. and M.P.; formal analysis, G.G. and M.P.; investigation, G.G. and M.P.; resources, M.P.; data curation, M.P.; writing—original draft preparation, M.P.; writing—review and editing, L.C. (Laura Carnevali), L.C. (Lorenzo Ciani), A.F., G.G. and M.P.; visualization, G.G. and M.P.; supervision, A.F.; project administration, A.F.; funding acquisition, A.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data sharing not applicable. No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CTMC	Continuous Time Markov Chain
DFT	Dynamic FT
DRBD	Dynamic RBD
FT	Fault Tree
FTRE	FT with Repeated Events
GSPN	Generalized Stochastic Petri Net
OS	Operating System
RBD	Reliability Block Diagram
RG	Reliability Graph
SAN	Stochastic Activity Network
SMP	Symmetric Multi-Processing
SPN	Stochastic Petri Net
SRN	Stochastic Reward Net
STPN	Stochastic Time Petri Net

References

1. ISO/IEC/IEEE. *International Standard-Systems and Software Engineering—Vocabulary*; ISO/IEC/IEEE 24765:2010(E); IEEE: New York, NY, USA, 2010; pp. 1–418. [CrossRef]
2. CENELEC. *EN 50126-1: Railway Applications–The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS)-Part 1: Generic RAMS Process*; Technical Report; CENELEC: Brussels, Belgium, 2017.
3. Trivedi, K.S.; Bobbio, A. *Reliability and Availability Engineering*; Cambridge University Press: Cambridge, UK, 2017. [CrossRef]
4. Mahboob, Q.; Zio, E. *Handbook of RAMS in Railway Systems: Theory and Practice*; CRC Press: Boca Raton, FL, USA, 2018. [CrossRef]
5. Moskowitz, F. The analysis of redundancy networks. *Trans. Am. Inst. Electr. Eng. Part I Commun. Electron.* **1958**, *77*, 627–632. [CrossRef]

6. IEC. *IEC 61078: Reliability Block Diagrams*; Technical Report; IEC: Geneva, Switzerland, 2016.
7. Hixenbaugh, A.F. *Fault Tree for Safety*; Technical Report; Boeing Aerospace Company: Seattle, WA, USA, 1968.
8. IEC. *IEC 61025: Fault Tree Analysis (FTA)*; Technical Report; IEC: Geneva, Switzerland, 2006.
9. Rubino, G. Network reliability evaluation. In *State-of-the-Art in Performance Modeling and Simulation*; Gordon & Breach Books: London, UK, 1998.
10. Bryant, R.E. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* **1986**, *C-35*, 677–691. [CrossRef]
11. Ericson, C. Fault Tree Analysis-A History. In Proceedings of the 17th International System Safety Conference, Orlando, FL, USA, 16–21 August 1999; pp. 1–9.
12. Stewart, W. *Introduction to the Numerical Solution of Markov Chains*; Princeton University Press: Princeton, NJ, USA, 1994.
13. IEC. *IEC 61165: Application of Markov Techniques*; Technical Report; IEC: Geneva, Switzerland, 2006.
14. Molloy, M. Performance Analysis Using Stochastic Petri Nets. *IEEE Trans. Comput.* **1982**, *31*, 913–917. [CrossRef]
15. Marsan, M.A.; Conte, G. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comput. Syst.* **1983**, *2*, 93–122. [CrossRef]
16. Vicario, E.; Sassoli, L.; Carnevali, L. Using stochastic state classes in quantitative evaluation of dense-time reactive systems. *IEEE Trans. Softw. Eng.* **2009**, *35*, 703–719. [CrossRef]
17. IEC. *IEC 62551: Analysis Techniques for Dependability—Petri Net Techniques*; Technical Report; IEC: Geneva, Switzerland, 2012.
18. Ciardo, G.; Blakemore, A.; Chimento, P.F.; Muppala, J.K.; Trivedi, K.S. Automated Generation and Analysis of Markov Reward Models Using Stochastic Reward Nets. In *Linear Algebra, Markov Chains, and Queueing Models*; Meyer, C.D., Plemmons, R.J., Eds.; Springer-Verlag: New York, NY, USA, 1993; pp. 145–191.
19. Ciardo, G.; Trivedi, K.S. A decomposition approach for stochastic reward net models. *Perform. Eval.* **1993**, *18*, 37–59. [CrossRef]
20. Meyer, J.; Movaghar, A.; Sanders, W. Stochastic Activity Networks: Structure, Behavior, and Application. In Proceedings of the International Workshop on Timed Petri Nets, Torino, Italy, 1–3 July 1985; pp. 106–115.
21. Sanders, W.H.; Meyer, J.F. Stochastic Activity Networks: Formal Definitions and Concepts. In *Lectures on Formal Methods and Performance Analysis: First EEF/Euro Summer School on Trends in Computer Science Bergen Dal, The Netherlands, 3–7 July 2000*; Brinksma, E., Eds.; Springer: Berlin/Heidelberg, Germany, 2000; pp. 315–343. [CrossRef]
22. Malhotra, M.; Trivedi, K.S. Power-hierarchy of dependability-model types. *IEEE Trans. Reliab.* **1994**, *43*, 493–502. [CrossRef]
23. Distefano, S.; Puliafito, A. Dynamic reliability block diagrams: Overview of a methodology. In Proceedings of the European Safety and Reliability Conference 2007, ESREL 2007-Risk, Reliability and Societal Safety, Stavanger, Norway, 25–27 June 2007; Volume 2.
24. Distefano, S.; Puliafito, A. Dependability Evaluation with Dynamic Reliability Block Diagrams and Dynamic Fault Trees. *IEEE Trans. Dependable Secur. Comput.* **2009**, *6*, 4–17. [CrossRef]
25. Dugan, J.B.; Bavuso, S.J.; Boyd, M.A. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Trans. Reliab.* **1992**, *41*, 363–377. [CrossRef]
26. Codetta-Raiteri, D. The Conversion of Dynamic Fault Trees to Stochastic Petri Nets, as a case of Graph Transformation. *Electron. Notes Theor. Comput. Sci.* **2005**, *127*, 45–60. [CrossRef]
27. Volk, M.; Weik, N.; Katoen, J.P.; Nießen, N. A DFT Modeling Approach for Infrastructure Reliability Analysis of Railway Station Areas. In *Formal Methods for Industrial Critical Systems*; Larsen, K.G., Willemse, T., Eds.; Springer International Publishing: Cham, Switzerland 2019; pp. 40–58.
28. Carnevali, L.; Ciani, L.; Fantechi, A.; Papini, M. A novel layered approach to evaluate reliability of complex systems. In Proceedings of the 2019 IEEE 5th International forum on Research and Technology for Society and Industry (RTSI), Florence, Italy, 9–12 September 2019; pp. 291–295. [CrossRef]
29. Papini, M. Reliability Evaluation of an Industrial System Through Predictive Diagnostics. Ph.D. Thesis, Universitá degli Studi di Firenze, Florence, Italy, 2021.
30. Iannino, A.; Musa, J.D. Software Reliability. In *Advances in Computers*; Marshall, C.Y., Ed.; Elsevier: Amsterdam, The Netherlands, 1990; Volume 30, pp. 85–170. [CrossRef]
31. Lyu, M. *Handbook of Software Reliability Engineering*; McGraw-Hill, Inc.: New York, NY, USA, 1996.
32. Mzyk, R.; Paszkiel, S. Influence of Program Architecture on Software Quality Attributes. In *Control, Computer Engineering and Neuroscience*; Paszkiel, S., Ed.; Springer International Publishing: Cham, Switzerland, 2021; pp. 322–329. [CrossRef]
33. Wood, A. Predicting software reliability. *Computer* **1996**, *29*, 69–77. [CrossRef]
34. Pham, H. *System Software Reliability (Springer Series in Reliability Engineering)*; Springer-Verlag: Berlin/Heidelberg, Germany, 2006. [CrossRef]
35. Ballerini, S.; Carnevali, L.; Paolieri, M.; Tadano, K.; Machida, F. Software rejuvenation impacts on a phased-mission system for Mars exploration. In Proceedings of the 2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Pasadena, CA, USA, 4–7 November 2013; pp. 275–280. [CrossRef]
36. Paolieri, M.; Biagi, M.; Carnevali, L.; Vicario, E. The ORIS Tool: Quantitative Evaluation of Non-Markovian Systems. *IEEE Trans. Softw. Eng.* **2019**, in press. [CrossRef]
37. RBDTool. Web Page. Available online: http://pages.mtu.edu/~pjbonamy/rbdtool.html (accessed on 13 November 2019).
38. Edraw Block Diagram. Web Page. Available online: https://www.edrawsoft.com/reliability-block-diagram-software.php (accessed on 23 April 2021).

39. Reliability Workbench. Web Page. Available online: https://www.isograph.com/software/reliability-workbench/rbd-analysis/ (accessed on 23 April 2021).
40. Relyence RBD. Web Page. Available online: https://www.relyence.com/products/rbd/ (accessed on 23 April 2021).
41. Sahner, R.A.; Trivedi, K.S.; Puliafito, A. *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*; Kluwer Academic Publishers: Alphen aan den Rijn, The Netherlands, 1996. [CrossRef]
42. SHARPE. Web Page. Available online: https://sharpe.pratt.duke.edu/ (accessed on 23 April 2021).
43. librbd. Web Page. Available online: https://github.com/marcopapini/librbd (accessed on 23 April 2021).
44. Siewiorek, D.P.; Swarz, R.S. *Reliable Computer Systems: Design and Evaluation*, 3rd ed.; A. K. Peters, Ltd.: Natick, MA, USA, 1998.
45. Catelani, M.; Ciani, L.; Venzi, M. RBD Model-Based Approach for Reliability Assessment in Complex Systems. *IEEE Syst. J.* **2019**, *13*, 2089–2097. [CrossRef]
46. Fourment, M.; Gillings, M. A comparison of common programming languages used in bioinformatics. *BMC Bioinform.* **2008**, *9*, 82. [CrossRef] [PubMed]
47. IEEE. IEEE Standard for Floating-Point Arithmetic. In *IEEE Std-754-2019 (Revision IEEE-754-2008)*; IEEE: New York, NY, USA, 2019; pp. 1–84.
48. IEEE. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX^{TM}) Base Specifications, Issue 7. In *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*; IEEE: New York, NY, USA, 2018; pp. 1–3951. [CrossRef]
49. pthreads-win32. Web Page. Available online: http://sourceware.org/pthreads-win32/ (accessed on 23 April 2021).
50. Cygwin. Web Page. Available online: https://www.cygwin.com/ (accessed on 23 April 2021).
51. Telcordia SR-332. *Reliability Prediction Procedure for Electronic Equipment*; Technical Report Issue 4; Telcordia Network Infrastructure Solutions (NIS): Bridgewater, NJ, USA, 2016.