Efficient Algorithms for Analyzing Cascading Failures in a Markovian Dependability Model

Mihir Sanghavi, Sashank Tadepalli, and Marvin K. Nakayama
Computer Science Department
New Jersey Institute of Technology
Newark, NJ 07102

Abstract

We develop an efficient algorithm to construct and solve a Markovian dependability system with cascading failures. Our algorithm reduces runtimes by orders of magnitude compared to a previous method. We also develop some heuristics based on the idea of most likely paths to failure to further reduce the computational efforts by instead constructing an approximate model. We present numerical results demonstrating the effectiveness of our approaches.

1 Introduction

Modern society relies on complex stochastic systems that operate in uncertain environments. These systems can suffer from cascading failures, in which the failure of one part of the system causes other parts to also fail. Examples include networks [10], electric power grids [8], national infrastructures [19], and transportation and communication systems [32]. Cascading failures in these systems can be catastrophic, causing widespread disruptions and damage.

In this paper we develop methods for analyzing a dependability system with cascading failures. We model such a system as a continuous-time Markov chain (CTMC; e.g., Chapter 5 of [24]), where the system is a collection of components operating in a randomly changing environment, where each component can fail and be repaired with specified failure and repair rates. We represent a cascading failure as a tree of probabilistically failing components that fail simultaneously, where the root of the tree is the failing component that *triggers* the cascade. The root probabilistically causes components from a specified set to fail simultaneously, with each component in the set having its own *component-affected probability*. Each of these secondary failures subsequently cause other components to probabilistically fail simultaneously, and so on. The *rate* of the resulting tree depends on the failure rate of the root and the component-affected probabilities of the failing and non- failing components in the cascade.

Analyzing such a Markovian dependability model with cascading failures presents significant challenges. Even simply building the infinitesimal generator matrix Q of the CTMC can be extremely time consuming, and indeed, the amount of work required can

grow exponentially in the number of components in the system. For example, consider a simple system with components A, B and C, where the failure of any one component can cause each of the others to fail simultaneously with certain probabilities. For a CTMC transition (x, y) in which all three components fail simultaneously, there are 9 corresponding cascading-failure trees: A causing B to fail, and then B causes C to fail; A causing B and C to simultaneously fail; B causing A causing C to fail; and so on. If the different components' failure rates and the component-affected probabilities differ, then each of the 9 trees has a different rate, and computing the (x, y)-entry in Q requires summing the rates of all 9 trees. In general, the number of trees corresponding to a single collection of components simultaneously failing can grow exponentially in the size of the set, with up to m^{m-1} different trees corresponding to a collection of m components failing in a cascade in the worst case [15]. Moreover, we need to examine each possible set of components that can simultaneously fail, and the number of such sets is exponential in the number of components in the system. Thus, just constructing the Q-matrix for the CTMC can require tremendous computational effort.

The paper [15] considers the same dependability model and develops an algorithm to generate the Q-matrix, but that algorithm there often requires enormous runtimes to build and solve larger models. We now develop new algorithms, which are significantly more efficient and can decrease the computational effort by orders of magnitude. We have developed software implementing our techniques, and we call the package the Dependability Evaluator of Cascading Failures (DECaF).

Previous tree-generation algorithms exist (e.g., Section 2.3.4.4 of [16]) for enumerating all possible trees in which there are no limitations on a node's possible children. But in our model, we restrict the children a particular node can have, which is why [15] and the current paper needed to develop new tree-generation algorithms.

In addition, since the time to construct the Q-matrix inherently grows exponentially because of the trees to build, we also propose heuristics to reduce the computational effort by only generating a subset of the trees. Our method exploits the idea of the *most likely paths to failure*, which has been previously utilized to design effective quick simulation methods based on importance sampling for analyzing highly dependable systems [13, 22]. We apply the concept to try to generate only the cascading-failure trees that arise on the most likely paths to failure, since these paths typically contribute most to the dependability measures computed, and leave out those trees on less likely paths. The omission leads to an approximate Q-matrix, which incurs errors in the resulting dependability measures. We explore the trade-off of the time savings from skipping trees with the error in the dependability measures, and numerical results show that our techniques can work well in practice.

The rest of the paper has the following layout. Section 2 reviews related work on dependability models, with a particular focus on cascading failures and other component interactions. We describe the model in Section 3. Section 3.1. Section 4 describes our new algorithms to build the CTMC's Q-matrix and presents numerical results comparing its runtime to that of the implementation in [15]. In Section 5 we develop our various heuristics that reduce runtime by not generating all trees at the expense of introducing error in the dependability measures, and we explore the trade-off through experiments. We apply our methods to a large cloud-computing model in Section 6, and Section 7 ends the paper with some concluding remarks.

2 Related Work

As mentioned in Section 1, the model we consider was previously studied in [15], but our new algorithms can solve large models with orders-of-magnitude reductions in runtime. The SAVE (System Availability Estimator) package [4], developed at IBM, analyzes a similar Markovian dependability model with cascading failures having the restriction that there is only one level of cascading; i.e., the root of a tree can cause other components to simultaneously fail, but those subsequent failures cannot cause further failures. Allowing for more than a single level of cascading makes the CTMC model we consider tremendously more difficult to construct.

Other modeling techniques, such as fault trees, reliability block diagrams (RBDs), and reliability graphs, have also been applied to study dependability systems, but these approaches do not allow for the level of details that are possible with CTMCs [21]. However, a notable drawback of CTMCs is the explosive growth in the size of the state space, which increases exponentially in the number of components in the system. Other packages for analyzing Markovian dependability models include SHARPE [25], SURF [7], SURF-2 [2], TANGRAM [3], and HIMAP [17].

Instead of assuming a Markovian system, some packages work with other mathematical models, such as stochastic Petri nets, which are analyzed by SNPN [14]. OpenSESAME [28] also solves stochastic Petri nets described via a high-level modeling language, which allows for cascading failures through failure dependency diagrams, but the complexity of the cascades that can be handled is not as great as in our model. The Galileo package [26] examines dynamic fault trees, which can model certain types of cascading failures via functional dependency gates; however, its modeling capability is limited. The paper [18] considers Bayesian networks for studying reliability systems.

Other mathematical modeling techniques that allow some forms of cascading failures or component interactions include Boolean driven Markov processes (BDMP) [5], common-cause and common-mode failures [1, 6], and coverage [11]. DRBD [30] uses dynamic reliability block diagrams, which extend traditional RBDs to allow for certain component interactions.

3 Model

We work with the stochastic model of [15], which considers the evolution over time of a repairable dependability system operating in a randomly changing environment. The system consists of a collection $\Omega = \{1, 2, \ldots, N\}$ of $N < \infty$ component types. Each component type $i \in \Omega$ has a redundancy $1 \leqslant r_i < \infty$, and the r_i components of type i are assumed to be identical. A component can be either operational (up) or failed (down).

The environment changes randomly within a set $\mathcal{E} = \{0, 1, 2, \ldots, L\}$. For example, we can think of the environment as representing the current load on the system, and if there are two possible environments, 0 and 1, then 0 (resp., 1) may represent a low (resp., high) load. Once the environment enters $e \in \mathcal{E}$, it remains there for an exponentially distributed amount of time with rate $\nu_e > 0$, after which the environment changes to e' with probability $\delta_{e,e'} \geqslant 0$, where $\delta_{e,e} = 0$ and $\sum_{e' \in \mathcal{E}} \delta_{e,e'} = 1$. We assume the matrix $\delta = (\delta_{e,e'} : e, e' \in \mathcal{E})$ is irreducible; i.e., for each $e, e' \in \mathcal{E}$, there exists $k \geqslant 1$ and a sequence $e_0 = e, e_1, e_2, \ldots, e_k = e'$ with each $e_i \in \mathcal{E}$ such that $\prod_{i=0}^{k-1} \delta_{e_i,e_{i+1}} > 0$.

In other words, it is possible to eventually move from environment e to environment e'.

The components in the system can randomly fail and then be repaired. When the environment is $e \in \mathcal{E}$, the failure rate and repair rate of each component of type i are $\lambda_{i,e} > 0$ and $\mu_{i,e} > 0$, respectively. If there is only one environment e, i.e., $|\mathcal{E}| = 1$, then the lifetimes and repair times of components of type i are exponentially distributed with rates $\lambda_{i,e}$ and $\mu_{i,e}$, respectively. Exponential distributions are frequently used to model lifetimes of hardware and software components; e.g., see [29]. We assume that all operating components of a type i have the same failure rate $\lambda_{i,e}$ in environment e. Thus, in a system with redundancies for which not all components of a type are needed for operation of the system, the extras are "hot spares" since they fail at the same rate as the main components.

Our model includes probabilistic instantaneous cascading failures, which occur as follows. The ordered set Γ_i specifies the types of components that a failure of a type-i component can cause to simultaneously fail. When a component of type i fails, it causes a single component of type $j \in \Gamma_i$ to fail at the same time with probability $\phi_{i,j}$ (if there are components of type j up), and we call $\phi_{i,j}$ a *component-affected probability*. The events that the individual components of types $j \in \Gamma_i$ fail immediately are independent. Thus, when a component of type i fails, there are independent "coin flips" to determine which components in Γ_i fail, where the coin flip for $j \in \Gamma_i$ comes up heads (one component of type j fails) with probability $\phi_{i,j}$ and tails (no components of type j fail) with probability $1 - \phi_{i,j}$.

We allow for a cascading failure to continue as long as there are still components operational in the system. For example, the failure of a component of type i may cause a component of type j to fail (with probability $\phi_{i,j}$), which in turn makes a component of type k fail (with probability $\phi_{j,k}$), and so on. As noted in [15], the SAVE package [4] allows for only one level of cascading, but the unlimited cascading in our model makes it significantly more difficult to analyze.

We can think of a cascading failure as a tree of simultaneously failing components. The root is the component, say of type i, whose failure *triggers* the cascade. The root's children, which are from Γ_i, are those components whose immediate failures were directly caused by the root's failure. At any non-root level of the tree, these components' failures were directly caused by the failure of their parents at the previous level. Although all the failing components in a cascade fail at the same time, we need to specify an order in which they fail for our problem to be well-defined, as we explain later in Section 3.2. We assume the components in a tree fail in breadth-first order.

There is a single repairman who fixes failed components using a processor- sharing discipline. Specifically, if the current environment is e and there is only one failed component, which is of type i, then the repairman fixes that component at rate $\mu_{i,e}$. If there are b components currently failed, then the repairman allocates $1/b$ of his effort to each failed component, so a failed component of type i is repaired at rate $\mu_{i,e}/b$.

3.1 Markov Chain

We want to analyze the behavior of the system as it evolves over time. Because of the processor-sharing repair discipline and the exponential rates for the event lifetimes, it will suffice to define the state of the system as a vector containing the number of failed components of each type and the current environment. Thus, let $S = \{x =$

$(x_1, x_2, \ldots, x_N, x_{N+1}) : 0 \leqslant x_i \leqslant r_i \ \forall i \in \Omega, \, x_{N+1} \in \mathcal{E}\}$ be the state space, and let $Z = [Z(t) : t \geqslant 0]$ be the continuous-time Markov time (CTMC) living on S keeping track of the current state of the system. (If we had instead considered a first-come-first-served repair discipline, then the state space would need to be augmented to keep track of the order in which current set of down components failed.) We assume that Z starts in environment $0 \in \mathcal{E}$ with no components failed, i.e., state $(0, 0, \ldots, 0)$. As noted in [15] the CTMC is irreducible and positive recurrent.

We now describe the CTMC's infinitesimal generator matrix $Q = (Q(x, y) : x, y \in S)$, where $Q(x, y)$ is the rate that the CTMC Z moves from state $x = (x_1, \ldots, x_N, x_{N+1})$ to state $y = (y_1, \ldots, y_N, y_{N+1})$. If $y_i = x_i$ for each $i \in \Omega$ and $y_{N+1} \neq x_{N+1}$, then (x, y) is an *environment transition* with $Q(x, y) = \nu_{x_{N+1}} \delta_{x_{N+1}, y_{N+1}}$. If $y_i = x_i - 1$ for one $i \in \Omega$, $y_j = x_j$ for each $j \in \Omega - \{i\}$, and $y_{N+1} = x_{N+1}$, then (x, y) is a *repair transition* corresponding to the repair of a component of type i, and $Q(x, y) = x_i \mu_{i, x_{N+1}} / (\sum_{j \in \Omega} x_j)$. If $y_i \geqslant x_i$ for all $i \in \Omega$ with $y_j > x_j$ for some $j \in \Omega$ and $y_{N+1} = x_{N+1}$, then (x, y) is a *failure transition* in which $y_i - x_i$ components of type i fail, $i \in \Omega$. Any other (x, y) with $x \neq y$ not falling into one of the above three categories is not possible, so $Q(x, y) = 0$. The diagonal entry $Q(x, x) = -\sum_{y \neq x} Q(x, y)$, as required for a CTMC; e.g., see Chapter 5 of [24].

We now determine the rate $Q(x, y)$ of a failure transition (x, y). First consider the case when cascading failures are not possible, i.e., $\Gamma_i = \varnothing$ for each type i. Then the only possible failure transitions (x, y) have $y_i = x_i + 1$ for one $i \in \Omega$, $y_j = x_j$ for each $j \in \Omega - \{i\}$, and $y_{N+1} = x_{N+1}$, and this corresponds to a single component of type i failing. Then $Q(x, y) = (r_i - x_i) \lambda_{i, x_{N+1}}$.

Cascading failures complicate the computation of $Q(x, y)$ for a failure transition (x, y). As mentioned before, a cascading failure is modeled as a tree T built from the multiset B of simultaneously failing components, where B has $y_\ell - x_\ell$ failing components of type ℓ, $\ell \in \Omega$. A tree T in a transition starting from a state x has a rate

$$R(T) \equiv R(T, x) = (r_i - x_i) \lambda_{i, x_{N+1}} \, \rho \, \eta, \tag{1}$$

where

- $(r_i - x_i) \lambda_{i, x_{N+1}}$ is the failure rate of the root (assumed here to be of type i),
- $\rho = \rho(T)$ is the product of the $\phi_{j,k}$ terms for a parent node of type j causing a child of type $k \in \Gamma_j$ to fail, and
- $\eta = \eta(T, x)$ is the product of the $1 - \phi_{j,k}$ terms from a node of type j *not* causing a component of type $k \in \Gamma_j$ to fail when there are components of type k up.

A difficulty arises since there can be many such trees corresponding to the multiset B of components failing in (x, y), and calculating $Q(x, y)$ requires summing $R(T)$ over all possible trees T that can be constructed from B. The number of such trees grows exponentially in the number of failing components in the cascade; see [15].

3.2 Example of Computing a Tree's Rate

We now provide an example of computing the rate $R(T)$ of a tree T. Let $\Omega = \{A, B, C\}$, with $r_A = r_B = r_C = 4$. Also, let $\Gamma_A = \{B, C\}$, $\Gamma_B = \{A, C\}$, and $\Gamma_C = \{A, B\}$. Suppose that $\mathcal{E} = \{0\}$, and consider the failure transition (x, y) with $x = (2, 2, 3, 0)$ and $y = (4, 4, 4, 0)$. Thus, (x, y) corresponds to 2 components each of types A and B

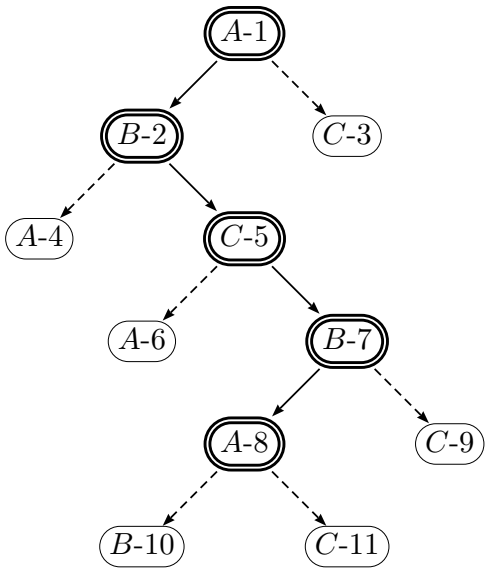

Figure 1: An example of a supertree.

failing and a single component of type C failing. One possible tree T corresponding to (x, y) is shown in Figure 1. We assume the nodes in T fail in breadth-first order.

The nodes depicted as double circles form the tree of failing components. The single circles correspond to components in some Γ_i but did not fail. A component type j in some Γ_i could have *not failed* because either there are components of type j up at this point but its coin flip came up tails (with probability $1 - \phi_{i,j}$), or there were no more components of type j up at this point. Each node has a label of the form i-ID, where i denotes the type of the component for that node, and ID is the position of the node in a breadth-first ordering of all the nodes (single and double circles). We include the IDs just to simplify the discussion here. We call the tree of all nodes the *supertree* corresponding to the tree T of failing nodes. The supertree is used to compute $R(T)$ of T as follows. Let u_i be the number of components of type i available in the system. Since the root is a component of type A and there are $u_A = r_A - x_A = 2$ components of type A at the start of the transition (x, y), the rate of the trigger of the cascade is $2\lambda_{A,0}$. The root then causes a component of type B to fail at node ID 2, and this occurs with probability $\phi_{A,B}$. The node at ID 3 did not fail, and at this point there are $u_C = r_C - x_C = 1 > 0$ components of type C still up, so this non-failure occurs with probability $1 - \phi_{A,C}$. Instead of stepping through the rest of the supertree one node at a time, we notice that all the $\phi_{i,j}$ terms must be included if T is to correspond to (x, y). Thus, in (1) we have $\rho = \phi_{A,B}\,\phi_{B,C}\,\phi_{C,B}\,\phi_{B,A}$. We notice the following when calculating η in (1):

- $1 - \phi_{i,j}$ terms are included if and only if there are still components of type j up.
- Each time we encounter a node of type j that has failed in the breadth- first enumeration of T, we decrement u_j by 1.

Keeping these observations in mind, we now calculate η. For component type A we have $u_A = 2$ before we traverse through T. As we do a breadth-first traversal through T, at

ID 1, $u_A = 1$. We see that $u_A > 0$ until ID 8. So η includes terms $1-\phi_{B,A}$ and $1-\phi_{C,A}$ from IDs 4 and 6, respectively. For component type B we have $u_B = r_B - x_B = 2$ before we traverse through T. Since components of type B have already been exhausted at ID 7, we do not include $1-\phi_{A,B}$ at ID 10 in η. For component type C we see that $u_C = 1$ before we traverse through T, and $u_C = 0$ at ID 5. Hence, the only contribution to η from a component of type C not failing is $1-\phi_{A,C}$ from ID 3. Taking the product over all component types yields $\eta = (1-\phi_{B,A})\,(1-\phi_{C,A})\,(1-\phi_{A,C})$. Therefore, $R(T)$ is $2\lambda_{A,0}\ \rho\ \eta$. In our implementation η is calculated through a data structure called *BFHist*, which we describe later in Section 4.1.

We previously stated that the order in which the components fail in a tree must be specified for the tree's rate to be well defined. To see why, suppose instead that the components in Figure 1 fail in depth-first order. The depth-first traversal of T is A-1, B-2, A-4, C-5, A-6, B-7, A-8, B-10, C-11, C-9, C-3. Initially, the number of components up of each type are $u_A = 2$, $u_B = 2$ and $u_C = 1$ as before. In this traversal a component of type C fails at ID 5, which makes $u_C = 0$.

Thus, η for the depth-first traversal does not include the terms $1-\phi_{A,C}$, $1-\phi_{B,C}$ and $1-\phi_{A,C}$ from the subsequent type-C nodes at IDs 11, 9 and 3. In contrast, the breadth-first traversal includes one $1-\phi_{A,C}$ term at ID 3. Hence, it is necessary to define the order in which the components fail, even though they fail simultaneously.

4 Algorithms

We now provide efficient algorithms for generating all possible trees and constructing the Q-matrix. A tree corresponds to a multiset of particular components failing, and cascading failures starting from different states can have the same multiset of components failing. Hence, a particular tree may correspond to several different transitions. Our algorithm generates each possible tree only once and determines all the transitions to which this tree corresponds. This avoids generating the same tree numerous times for each corresponding transition, as was originally done in [15]. Moreover, rather than building each new tree from scratch, as was done in [15], our current algorithm builds larger trees from smaller ones already considered, leading to additional savings in the overall computational effort.

Computing the rate (1) of a given tree depends on the state from which the cascading failure began and the multiset of components that fail in the cascade. We do not actually construct the supertree in our algorithm to compute a tree's rate, but instead build a *breadth-first history* to keep track of the information necessary to compute η in (1). This breadth-first history concisely recounts the creation of the tree and thus allows us to obtain η without having to build supertrees per transition as was done in [15]. This is all done in Algorithms 1 (SeedTrees), 2 (AddTreeLevel) and 3 (ComputeTreeRate).

SeedTrees starts the tree generation and initializes the necessary data structures for AddTreeLevel. AddTreeLevel adds a new level to an existing tree in a recursive fashion, updates ρ in (1) to include the component-affected probabilities, and builds the tree's breadth- first history. ComputeTreeRate computes the rate of a completed tree for all the transitions it corresponds to using ρ and η computed from the breadth-first history populated in AddTreeLevel. We will now discuss each algorithm in detail. References to line numbers in the algorithms are given within angled brackets $\langle\ \rangle$.

4.1 SeedTrees

SeedTrees loops through all component types to choose a root for the tree to be constructed $\langle 1 \rangle$. One of the data structures initialized in SeedTrees is *BFHist*, which stores the breadth-first history using an array of linked lists, where the array is indexed by component type. Each linked list stores the respective parents of nodes exclusive to the supertree and stores the symbol @ for the nodes that *did* fail in the tree.

The dynamic array *level* contains the current bottom level of the tree that we are building, and it initially contains the root, which is of type *rootC.type* $\langle 5 \rangle$. We add the symbol @ to the list at *BFHist*[*rootC.type*] to denote that one component of the root's type has failed. We update *nFailed* by setting the counter for *rootC.type* to 1 $\langle 5-7 \rangle$. We initialize ρ in (1) to 1 since no components so far have been caused to fail by the failure of other components $\langle 8 \rangle$.

If the component type *rootC.type* at the root cannot cause any other components to fail, only the trivial tree of one node can be made with this type of root. We then evaluate this single-node tree's rate in ComputeTreeRate because it cannot be grown further. If $\Gamma_{rootC.type} \neq \varnothing$ (i.e., it can cause other types of components to fail), then we call AddTreeLevel to proceed with building taller trees by adding another level to the current tree $\langle 9-13 \rangle$.

Algorithm 1 SeedTrees(Γ)

where Γ is an array of ordered sets that describes which components can cause which other components to fail

```
1: for rootC.type ∈ Ω do
2:    level = [ ]; {Dynamic array of failed components at tree's current bottom level}
3:    nFailed = [0, 0, ..., 0]; {Array that counts failed components of each type in the
      tree}
4:    BFHist = [( ), ( ), ..., ( )]; {Array of linked lists that keeps a history of parent
      component types in breadth-first order; BFHist is indexed by component types}
5:    add rootC.type to level;
6:    nFailed[rootC.type] = 1;
7:    add @ to BFHist[rootC.type]; {Signifies one component of type rootC.type has
      failed}
8:    ρ = 1; {initialize product of component-affected probabilities}
9:    if Γ_rootC.type == ∅ then
10:      ComputeTreeRate(nFailed, BFHist, ρ, rootC.type);
11:   else
12:      AddTreeLevel(level, nFailed, BFHist, ρ, rootC.type);
13:   end if
14: end for
```

4.2 AddTreeLevel

Given the bottom-most tree level, AddTreeLevel determines all the possibilities for the next level that can be added to the current tree by taking the Cartesian product of

the power sets of the nonempty Γ_i for each of the failed components at the current bottom level. For each possible next level, we recursively call AddTreeLevel. We use $\mathcal{P}$ to denote the power set-like operation on an ordered set. In our realization of the power-set operation, components in each subset maintain their relative ordering from the original set. Note that by only maintaining one bottom-most level at a time, as opposed to passing the entire tree per recursive call, we save some memory.

We implement this power set-like operation by generating all possible binary numbers with $\sum_{i=1}^{|level|} \Gamma_{level[i]}$ bits. A total of $\prod_{i=1}^{|level|} 2^{\Gamma_{level[i]}}$ such binary numbers are generated. In the binary number, 1 denotes a failed node and 0 denotes a node that is included in a Γ set but did not fail. If we have a tree where none of the leaf nodes at the current bottom level can cause any other components to fail (i.e., have empty Γ_is), we get no nextLevelPossibilities $\langle\, 1 \,\rangle$.

Otherwise, we choose one possible choice for the failed components in the next level from the nextLevelPossibilities $\langle\, 2 \,\rangle$. Each node in T has three attributes: a component type i; an ID, which is its breadth-first ID in the supertree; and a parentID, which is the breadth-first ID of this node's parent in the supertree. For each node *parentC* in the current bottom level that acts as a parent, potentially causing other nodes to fail, we iterate through all of its possible children, i.e., through $\Gamma_{parentC.type}$. If any of these children actually fail, then they will be members of the set *oneNextLevelPossibility* $\langle\, 4-6 \,\rangle$. Now if the redundancy of the component type we just tried to add as a child has already been exhausted, it cannot fail and hence our tree is invalid and we move on to the next possibility $\langle\, 10-12 \,\rangle$.

If a child of type *childC.type* has failed (i.e., it exists within nextLevelPossibilities), the relevant data structures are updated to signify this occurrence. We increment the number of failed components of type *childC.type*, add the symbol @ to *BFHist*[*childC.type*] (to mark that a failure of this type has occurred at the current location in the tree), and update ρ by multiplying it by $\phi_{i,j}$ $\langle\, 11-13 \,\rangle$. If the child does not actually fail, then we add *parentC.type* to *BFHist*[*childC.type*]. Later in ComputeTreeRate for each transition (x, y), we evaluate which of these children could have failed but did not, and which could *not* have failed because their type's redundancy was exhausted. $\langle\, 14-16 \,\rangle$.

We do the above updates to the data structures for each potential parent node in *level* and each of its children in *oneNextLevelPossibility*. If at least one child has been added, we recursively add another level to the current tree $\langle\, 19-23 \,\rangle$. Otherwise, this tree has not changed in this pass through AddTreeLevel and we call ComputeTreeRate to compute the rate of the finalized tree $\langle\, 24-26 \,\rangle$. We make sure that trees are not double counted because once a tree passes through AddTreeLevel unmodified, it is processed and discarded. We avoid making duplicate trees because each oneNextLevelPossibility is unique.

Algorithm 2 AddTreeLevel(*level*, *nFailed*, *BFHist*, ρ, *rootC.type*)

where *level* is the current level of failed components,
nFailed counts failed components by type in the tree,
BFHist is breadth-first history,
ρ is a cumulative product of component-affected probabilities,
rootC.type is the root component's type in the current tree

1: $nextLevelPossibilities = \underset{\substack{i=1: \\ \Gamma_{level[i]} \neq \varnothing}}{\overset{|level|}{\times}} \mathcal{P}(\Gamma_{level[i]})$;
{Builds all possibilities for the next level (given the current level) by taking a Cartesian product of the power sets of non-empty Γ sets of failed nodes in the current level}
2: **for** *oneNextLevelPossibility* $\in$ *nextLevelPossibilities* **do**
3: *addedAChildFlag* = False;
4: **for** *parentC* $\in$ *level* **do**
5: **for** $i \in \Gamma_{parentC.type}$ **do**
6: **if** $\exists$ *childC* $\in$ *oneNextLevelPossibility* : *childC.type* == i && *childC.parentID* == *parentC.ID* **then**
7: *addedAChildFlag* = True;
8: **if** *nFailed*[*childC.type*] == $r_{childC.type}$ **then**
9: goto line 2; {Invalid Tree, it requires more components than available}
10: **end if**
11: *nFailed*[*childC.type*] = *nFailed*[*childC.type*] + 1;
12: add @ to *BFHist*[*childC.type*]; {One component of type *childC.type* has failed}
13: $\rho = \rho * \phi_{parentC.type,\, childC.type}$;
{Update rate with appropriate component-affected probability}
14: **else**
15: add *parentC.type* to *BFHist*[*childC.type*]; {One component of type *childC.type* has not failed, but was present in $\Gamma_{parentC.type}$}
16: **end if**
17: **end for**
18: **end for**
19: **if** *addedAChildFlag* **then**
20: AddTreeLevel(*oneNextLevelPossibility*, *nFailed*, *BFHist*, ρ, *rootC.type*);
{Tree can be grown further}
21: **else**
22: ComputeTreeRate(*nFailed*, *BFHist*, ρ, *rootC.type*);
{Current tree is complete because it cannot be grown further}
23: **end if**
24: **end for**

4.3 ComputeTreeRate

In ComputeTreeRate, for a given tree, we determine all of the transitions (x, y) in the Q-matrix that use this tree, and then update the total rates of each of those transitions (x, y) by adding in the rate of the current tree to the current rate for (x, y). However, even if several transitions use the same tree, the rate of the tree may differ for those transitions since the failure rate of the root and η are transition-dependent.

Let $x' = (x_1, x_2, \ldots, x_N)$ and $S' = \{(x_1, x_2, \ldots, x_N) : 0 \leqslant x_i \leqslant r_i, \forall i \in \Omega\}$. In line 1 of ComputeTreeRate, we only loop through $x' \in S'$ and not $x \in S$ because the

structure of a tree does not depend on the environment; only the tree's rate does. Also, as we explained in the example tree-rate calculation in Section 3.2, computing η in (1) requires keeping track of the number u_i of components of each type i still up as we perform a breadth-first traversal through the supertree.

Each time we encounter the symbol @ in the *BFHist*, we reduce the number available by one $\langle 5-7 \rangle$. If there are components still available, we update η $\langle 8-9 \rangle$. As soon as u_i reaches zero, we break because that type has then been exhausted and can no longer fail $\langle 10-12 \rangle$.

Once we have finished calculating η, we loop through all environments since the same tree can be used for transitions occurring in different environments $\langle 15 \rangle$. We convert $x' \in S'$ into a state $x \in S$ by appending an environment e $\langle 16 \rangle$. We generate a potential state y by adding *nFailed* to x' and appending environment e $\langle 17 \rangle$. Invalid states y will be generated in some instances because simply adding *nFailed* will cause some component types' number failed to exceed their redundancies $\langle 18-20 \rangle$. We finally compute the rate of the tree for the transition (x, y) using (1), and add this to the current cumulative rate $Q(x, y)$ for the transition $\langle 21 \rangle$.

Algorithm 3 ComputeTreeRate(*nFailed*, *BFHist*, ρ, *rootC.type*)

where *level* is the current bottom level of failed components,
nFailed counts the failed components of each type in the tree,
BFHist is breadth-first history,
ρ is a cumulative product of component-affected probabilities,
rootC.type is the root's type in the tree

```
1:  for x' ∈ S' do
2:     η = 1; {Cumulative product of complement probabilities of components that
       could have failed but did not}
3:     for i ∈ Ω do
4:        u_i = r_i − x'_i;
5:        for parentC.type in BFHist[i] do
6:           if parentC.type == @ then
7:              u_i = u_i − 1;
8:           else if u_i > 0 then
9:              η = η * (1 − φ_{parentC.type, i});
10:          else
11:             break; {need u_i > 0 or else there cannot be any more failed nodes of type
                i}
12:          end if
13:       end for
14:    end for
15:    for e ∈ E do
16:       x = (x', e);
17:       y = (x' + nFailed, e);
18:       if y is not a valid state then
19:          continue;
20:       end if
21:       Q(x, y) = (r_{rootC.type} − x[rootC.type]) * λ_{rootC.type,e} * ρ * η;
```

22: **end for**
23: **end for**

4.4 Computing a Tree's Rate Revisited

We now use our example in Section 3.2 to show how to compute η from (1) using the data structure *BFHist* for Figure 1. We assume the tree in Figure 1 was first constructed from several recursive calls to AddTreeLevel, which also built *BFHist* as follows.

A	@-1	B-2	C-5	@-8
B	@-2	@-7	A-8	
C	A-1	@-5	B-7	A-8

We then proceed to ComputeTreeRate to compute η. As in Section 3.2, prior to iterating through *BFHist*, we have $u_A = 2$, $u_B = 2$ and $u_C = 1$. Iteration through *BFHist* occurs as specified in ComputeTreeRate $\langle 3 - 14 \rangle$.

- Iteration through the linked list at index A is as follows: The @-1 means a component of type A has failed at ID 1, so we then decrement u_A to 1. Then for the next two entries, since u_A is still positive, η includes $(1-\phi_{B,A})$ and $(1-\phi_{C,A})$ from B-2 and C-5 respectively in its product. Finally @-8 means a component of type A failed at ID 8, so we decrease u_A to 0.
- Iteration through the linked list at index B is as follows: @-2 and @-7 mean components of type B have failed at IDs 2 and 7 respectively, so we decrement u_B from 2 to 1 and then from 1 to 0. Since u_B is now 0, we cannot include any terms from this row in η.
- Iteration through the linked list at index C is as follows: Since u_C starts out positive, we include $(1 - \phi_{A,C})$ from A-1. @-5 means a component of type C has failed at ID 5, so we decrement u_C to 0. Since $u_C = 0$, again we cannot include any terms from this row in η.

Multiplying the contributions from each index results in $\eta = (1 - \phi_{B,A})(1 - \phi_{C,A})(1 - \phi_{A,C})$.

4.5 Dependability Measures

Once the Q-matrix has been constructed, we can use it to compute various dependability measures. We first partition the state space $S = U \cup F$, where U (resp., F) is the set of states for which the system is operational (resp., failed). The partition is determined by a model specification giving conditions under which the system is considered to be operational; e.g., at least υ_i components of type i are up for each type i. One dependability measure is the *steady-state unavailability* (SSU), which we define as follows. Let $\pi = (\pi(x) : x \in S)$ be the nonnegative row vector defined such that $\pi Q = 0$ and $\pi e = 0$, where e is the column vector of all 1s; i.e., π is the steady-state probability vector of the CTMC; e.g., see Chapter 5 of [24]. The vector π exists and is unique since, as shown in [15], our CTMC is irreducible and positive recurrent. We

States	Comp. Types	Redundancies	Component-Affected Sets	Env.
81	A, B, C, D	$r_A = 2$, $r_B = 2$, $r_C = 2$, $r_D = 2$	$\Gamma_A = \{B, C\}$, $\Gamma_B = \{A, D\}$, $\Gamma_C = \varnothing$, $\Gamma_D = \{A, B, C\}$	1
288	A, B, C, D	$r_A = 3$, $r_B = 3$, $r_C = 2$, $r_D = 2$	$\Gamma_A = \{B, C\}$, $\Gamma_B = \{A, C\}$, $\Gamma_C = \{B, D\}$, $\Gamma_D = \{C\}$	2
640	A, B, C, D	$r_A = 4$, $r_B = 3$, $r_C = 3$, $r_D = 3$	$\Gamma_A = \{B, C\}$, $\Gamma_B = \{A, C\}$, $\Gamma_C = \{B, D\}$, $\Gamma_D = \{B\}$	2
125	A, B, C	$r_A = 4$, $r_B = 4$, $r_C = 4$	$\Gamma_A = \{B, C\}$, $\Gamma_B = \{A, C\}$, $\Gamma_C = \{A, B\}$	1
1944	A, B, C, D, E, F	$r_A = 2$, $r_B = 2$, $r_C = 2$, $r_D = 3$, $r_E = 2$, $r_F = 2$	$\Gamma_A = \Gamma_B = \{C, D\}$, $\Gamma_C = \{A, E\}$, $\Gamma_D = \{B, F\}$, $\Gamma_E = \Gamma_F = \varnothing$	2

Table 1: Description of the various models we used to analyze our algorithm

then define the SSU as $\sum_{x \in F} \pi(x)$, which is the long-run fraction of time the CTMC spends in F.

Another dependability measure is the *mean time to failure* (MTTF), which can be defined as follows. Let x_* denote the state in which all components are operational and the environment is $0 \in \mathcal{E}$; we previously assumed that the CTMC Z starts in state x_*. Define $T_F = \inf\{t > 0 : Z(t) \in F\}$, so the MTTF is $E[\, T_F \,|\, Z(0) = x_* \,]$, which we can compute in the following manner. Define the transition probability matrix $P = (P(x, y) : x, y \in S)$ of the embedded discrete-time Markov chain (DTMC) with $P(x, y) = -Q(x, y)/Q(x, x)$ for $x \neq y$, and $P(x, x) = 0$ (Chapter 5 of [24]). Also define the $|U| \times |U|$ matrix $P_U = (P(x, y) : x, y \in U)$ and $|U| \times |U|$ identity matrix I, and let $h = (h(x) : x \in U)$ be the column vector such that $h(x) = -1/Q(x, x)$, which is the expected holding time that the CTMC spends in state x in each visit there. Then let $m = (I - P_U)^{-1} h$, and the MTTF equals $m(x_*)$; e.g., see Section 7.9 of [27].

4.6 Comparison of Runtimes

We now compare the runtimes of the original version of the code [15] with our current implementation, as described in Section 4. We carry out the comparison on a set of different models described in Table 1, which gives for each model the number states in the state space S of the CTMC, the set of component types, the redundancies of each component type, the component-affected sets Γ_i, and the number of environments. In the text below we refer to each model by its number of states, e.g., the "125-state model." Note that the number of trees does not always grow as the number of states increases, but rather the relationships among the Γ sets and the component redundancies determine the amount of cascading possible.

Table 2 gives the computation times for various parts of the overall algorithms of the original code [15] and the current implementation. We compare the two implementations in terms of the tree- generation time (TGT) and Q-matrix processing time (QPT), which is the total time to build the Q-matrix, including building the trees. Thus, QPT also includes filling in the rates for the repair and environment transitions. For the current implementation, we also give the Q-matrix solving time (QST), which

States	Trees	Previous Version		New Version		
		TGT	QPT	TGT	QPT	QST
81	978	1.30	1.60	0.12	0.22	0.08
288	4507	19.00	19.62	0.26	0.36	0.16
640	27746	137.29	143.27	1.56	1.67	0.61
125	321372	114.25	114.58	6.01	6.11	0.09
1944	6328	6124.01	6358.85	1.35	1.51	8.27

Table 2: Number of trees, tree-generation time, Q-matrix processing time, and Q-matrix solving time across several models and the previous and new versions of the code.

includes the time to compute the MTTF and SSU, as described in Section 4.5. While the MTTF and SSU computations are performed using the OJAlgo package [23], the solving of dependability measures once the Q-matrix has been constructed is not the focus of our paper, and we can swap out the current solver with any other appropriate code.

Table 2 shows the enormous increases in efficiency that we get from the new version of the code. The current implementation decreases the TGT and QPT by about one order of magnitude on the 81-state model and by over a factor of 4200 on the 1944-state model. The efficiency gains are due to the changes described in the first paragraph of Section 4, as well as other improvements in the design of the algorithms and data structures discussed in Sections 4.1–4.4. For the new version of the code, the TGT mainly grows as a function of the number of trees.

5 Not Generating All Trees and the Resulting Error

The number of trees can grow exponentially in the number of components in the cascade [15], and this limits the size of the models that our algorithms in Section 4 can handle. To address this issue, we explored heuristic approaches that reduce the computational effort by selectively generating only certain trees. trees. We implement this by inserting the following extra code immediately after line ⟨ 18 ⟩ of AddTreeLevel (Algorithm 2) to halt building trees based on a stopping criterion.

```
18.1:  if stopping criterion satisfied then
18.2:     goto line 2;
18.3:  end if
```

This causes the algorithm to skip over to the next enumeration of the bottom- most tree level, so we do not generate certain trees; the omitted trees' rates are not computed and are not included in the Q-matrix. This saves computation time, but the resulting matrix Q' (that sums the rates of trees that are generated) can differ from the matrix Q that includes all trees. Solving for the MTTF and SSU with Q' rather than Q leads to inaccuracies in the values for these dependability measures. We investigated the trade-off in the time saved by halting tree generation versus the resulting error in the dependability measures computed from Q' instead of Q.

In designing a stopping criterion, we want to allow trees with large rates to be generated, since these often have a larger impact on the MTTF and SSU, and skip trees with small rates. We considered three criteria based on different types of thresholds.

We first consider a *height threshold*, and the stopping criterion is $height > \tau_{\rm h}$, where the global variable *height* keeps track of the current tree height and $\tau_{\rm h}$ is the threshold. Our implementation requires a slight change to AddTreeLevel, where we introduce *height* as a method parameter. We also modified line $\langle\, 20 \,\rangle$ to increment *height* on every successive recursive call.

We also consider a *node threshold* $\tau_{\rm n}$ in the stopping criterion $\sum_{i\in\Omega} nFailed[i] > \tau_{\rm n}$. constructed trees will contain a maximum of $\tau_{\rm n}$ failed components.

For the third stopping criterion, we use a *rate threshold* to stop tree generation when the tree rates get too small. Define

$$R'(T) = \bar{\lambda}_i \rho \tag{2}$$

for a tree T, where $\bar{\lambda}_i = \max_{e\in\mathcal{E}} \lambda_{i,e}$ is the maximum failure rate for the type i of the root of the tree T over all the different environments, and ρ is as defined in (1). Note that $R'(T)$ differs from the tree rate $R(T)$ in (1) since $R'(T)$ omits both η, which is the product of the $1 - \phi_{i,j}$ terms for components of types j that did not fail in the cascade but could have (because $j \in \Gamma_i$ of a component of type i that did fail), and the current number of up components of the root type. Also, $R'(T)$ uses the maximum failure rate $\bar{\lambda}_i$ of the root type i instead of the environment-specific failure rate. Then the stopping criterion is $R'(T) < \tau_{\rm r}$, where $\tau_{\rm r}$ is the specified threshold.

For each of the three stopping criteria, once a given tree T satisfies the criterion, the current tree will not be grown any further. This is clear for the node and height thresholds. For the rate threshold, adding additional nodes to T decreases ρ in (2) since it is multiplied by additional factors $\phi_{j,k} \leqslant 1$, so once the rate stopping criterion is satisfied, T will not be further enlarged.

Increasing $\tau_{\rm h}$ and $\tau_{\rm n}$ leads to a monotonic increase in the number of generated trees, whereas the number of generated trees decreases monotonically in $\tau_{\rm r}$. Setting $\tau_{\rm h} = \tau_{\rm n} = \infty$ or $\tau_{\rm r} = 0$ results in generating all trees, so the computed Q-matrix is exact and there is then no error in the computed MTTF and SSU.

We now present numerical results when applying one stopping criterion at a time for the 125-state model, where we set the failure rates $\lambda_{A,1} = \lambda_{B,1} = 0.02$, $\lambda_{C,1} = 0.01$, and component-affect probabilities $\phi_{A,B} = \phi_{B,C} = 0.2$, $\phi_{C,A} = \phi_{B,A} = 0.3$, $\phi_{A,C} = \phi_{C,B} = 0.4$. Also, the repair rates $\mu_{i,1} = 1$ for all types i, and the system is operational as long as at least 1 component is up of each type. Figure 2 shows the relative error of the MTTF and SSU (relative to the values when all trees are generated) as a function of the relative number of generated trees. As we expect, for each threshold, the error decreases as the number of trees increases. Also, for a fixed number of trees generated, the stopping criterion based on the rate threshold leads to smaller error; similarly, if we fix a level of error, the rate criterion requires fewer trees than the other two criteria to achieve that error. Hence, the points from using the rate threshold define the *efficient frontier*, analogous to the idea introduced by [20] in the context of financial portfolio selection. (The results for the other models in Table 1 are qualitatively similar.)

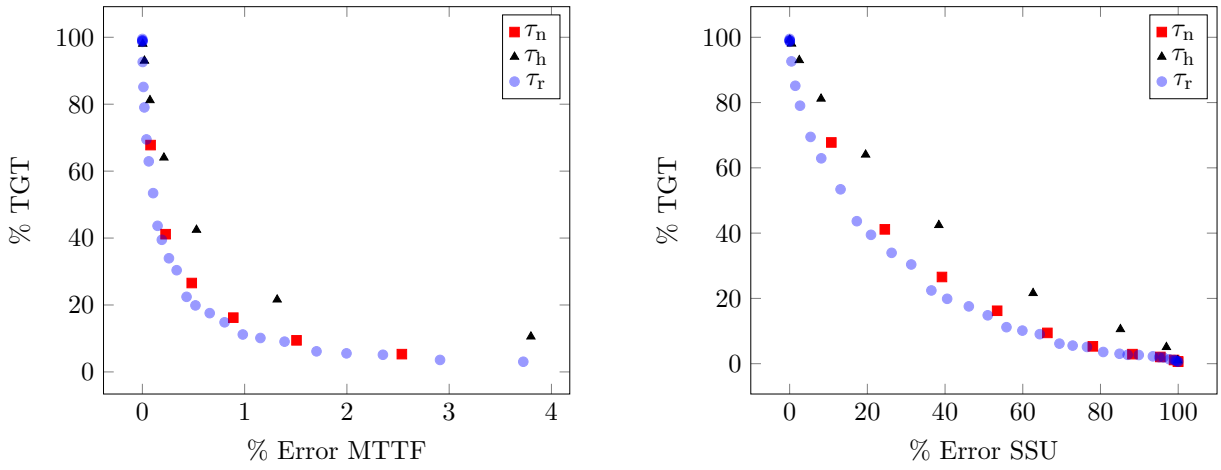

Figure 2: %TGT versus % Error in MTTF and SSU when iterating each threshold τ

5.1 Choosing an appropriate τ_r

The previous discussion shows that a stopping criterion based on a rate threshold appears to outperform the other thresholds we considered. We now discuss a heuristic that uses only the building blocks of the model to try to select an appropriate value for the rate threshold τ_r so that only a relatively small number of trees are generated but the resulting error in the dependability measures is small. The method exploits the idea that trees that occur in the most likely way the system fails are probably the ones whose rates contribute most to the computed dependability measures. Exactly identifying these trees is complicated, so we instead use various approximations and simplifying assumptions to roughly determine a value for τ_r that allows such trees to be built while precluding trees on less likely paths to failure.

We first give an overview of our approach. Assume that the system consists of highly reliable components [13] in the sense that the component failure rates are much smaller than the repair rates. Suppose the system-operational conditions require that at least υ_i components of each type i are up for the system to be operational, so the system is failed when at least $d_i = r_i - \upsilon_i + 1$ components have failed for some type i. We will focus on sequences of states (i.e., paths of the embedded DTMC) for which the first state in the sequence has all components up, the last state in the sequence is a failed state (i.e., in F), all states in between are operational (i.e., in U), and each successive pair of states is a failure transition (possibly a cascade). Such a sequence of states is a path to system failure, failing single cascade is usually when exactly d_i components of some type i fail (along with possibly other component types failing in cascades) along the path. For each component type i and each $1 \leqslant k \leqslant d_i$, we examine paths consisting of exactly k (failure) transitions (possibly some being cascades) in which a total of d_i components of type i fail. For each failure transition in the path, we only consider one tree (possibly with just a single node), even if there are multiple trees corresponding to that transition. If there is more than one tree corresponding to a particular transition in the path, we try to determine the tree that has the largest rate. rough approximation for the probability of the transition for the embedded DTMC. The product of the approximate transition probabilities along the path then gives an

approximate probability of the entire path. The path that maximizes the approximate path probabilities over all types i and numbers k of transitions in the path then provides an approximation to the most likely path to failure. Finally we set the rate threshold as the smallest rate of a tree along the approximate most likely path to failure.

We now provide details of the approach. First fix a component type i and a number k, $1 \leqslant k \leqslant d_i$, of transitions in a path to failure in which exactly d_i components of type i fail. We consider a path of the embedded DTMC having a succession of k cascading-failure trees, each with $f_i \equiv d_i/k$ components of type i (and possibly other component types) failing, where we initially assume f_i is an integer. For each of the k transitions along the path, we now want to approximate the structure of the most likely (i.e., highest-rate) tree T with f_i components of type i (and possibly other component types) failing. If $f_i = 1$, then the tree is just a single node of type i. (When d_i/k is not an integer, we first allocate $\lfloor d_i/k \rfloor$ failing components of type i to each of the k transitions, where $\lfloor \cdot \rfloor$ denotes the floor function. Then for the remaining $b_i \equiv d_i - k\lfloor d_i/k \rfloor$ failing components of type i, we allocate one additional failing component of type i to b_i of the transitions. Thus, b_i of the transitions along the path each have $\lfloor d_i/k \rfloor + 1$ type-i components failing, and the other $k - b_i$ transitions each have $\lfloor d_i/k \rfloor$ type-i failures. In this case, we let f_i be either $\lfloor d_i/k \rfloor$ or $\lfloor d_i/k \rfloor + 1$ in the discussion below.)

For any tree T, recall $R'(T)$ in (2) is the product of the maximum failure rate of the root and the product of the component-affected probabilities $\phi_{j,l}$ of components that fail in the cascade. Since adding more nodes to a tree will multiply its rate by additional $\phi_{j,l}$ terms, each of which is no greater than 1, a tree T with large $R'(T)$ will typically have not too many nodes and the $\phi_{j,l}$ terms included in ρ from (1) will often be relatively large. We equivalently try to find such a tree T with large $\ln(R'(T))$, which converts the product $R'(T)$ into a sum of logs. This transformation allows us to use a shortest-path algorithm on an appropriately defined graph to help identify such a tree.

Specifically, construct a weighted graph $G = (V, E, W)$, where $V = \Omega$ is the set of vertices, $E = \{(j, l) : l \in \Gamma_j\}$ is its set of edges, and $W = \{w_{j,l} : (j, l) \in E\}$ is the set of weights (costs), with $w_{j,l} = -\ln \phi_{j,l}$. Because $\ln \phi_{j,l} \leqslant 0$, large $\phi_{j,l}$ corresponds to small $w_{j,l}$. To try to identify a tree T with large $\ln(R'(T))$ and with exactly f_i type-i components failing, we will build it by first finding the lowest-cost cycle in G starting and ending in node i. This cycle corresponds roughly to the most likely way a type-i component triggers a cascade in which another type-i component fails. Then repeat the cycle enough times so that exactly f_i type-i components fail, thereby resulting in a tree for which $\ln(R'(T))$ should be large.

To find the lowest-cost cycle, we first use Dijkstra's algorithm (Section 24.3 of [9]) to determine the minimum-cost path in the weighted graph G from vertex j to vertex i for each $j \in \Gamma_i$, and let $c_{j,i}$ be the path's cost. Let $c_i = \min_{j \in \Gamma_i}(w_{i,j} + c_{j,i})$, which is the cost of the minimum-cost cycle in G from vertex i back to itself. Let $g_i = (g_{i,1}, g_{i,2}, \ldots, g_{i,l_i})$ be the sequence of nodes on this minimum-cost cycle, where $g_{i,1} = g_{i,l_i} = i$ and l_i is the number of vertices in the cycle (including i twice).

Using the minimum-cost cycle g_i, we then build a tree T_i that is a chain having exactly f_i components of type i failing by repeating this minimum-cost cycle $f_i - 1$ times. Specifically, let $g'_i = (g_{i,2}, g_{i,3}, \ldots, g_{i,l_i})$, which is the minimum-cost cycle g_i with the starting vertex omitted. Then T_i is a chain with root of type i, to which g'_i is concatenated $f_i - 1$ times. has exactly of other types. We use T_i as our approximation

of the tree corresponding to the most likely cascading-failure tree triggered by a failure of a component of type i and in which exactly f_i components fail. Recall that this tree is for one of the transitions along the path having k transitions. For this tree T_i, we can compute $R'(T_{i_0})$ in (2) as $\bar{\lambda}_i a_i^{f_i-1}$, where $a_i = \prod_{j=1}^{l_i-1} \phi_{g_{i,j},g_{i,j+1}}$, which is the product of component-affected probabilities along the minimum- cost cycle.

Now we build our approximation to the most likely path to failure consisting of exactly k transitions with a total of d_i components failing such that each transition (x, y) along the path has a tree T_i as constructed above. We also choose the environment in each state along the path to be the same $e_0 \in \mathcal{E}$, where e_0 is such that $\lambda_{i,e_0} = \max_{e\in\mathcal{E}} \lambda_{i,e}$. Now for each (x, y), we compute an approximate DTMC transition probability as

$$P'(x, y) = \frac{R(T_i, x)}{\sum_{j\in\Omega}(r_j - x_j)\lambda_{j,e_0} + \sum_{j\in\Omega} x_j\mu_{j,e_0}/(\sum_{l\in\Omega} x_l)},$$

where the numerator is the (exact) rate for the tree T_i from (1) and the denominator is the total rate out of state x. Then define $\zeta_{i,k}$ as the approximate probability of the entire path, which is the product of the $P'(x, y)$ for the transitions (x, y) along the path. Maximizing $\zeta_{i,k}$ over all types i and path lengths $1 \leqslant k \leqslant d_i$ leads to our approximation of the most likely path to failure.

Suppose that this approximate most likely path to failure corresponds to type i_0 and path length k_0. For this path, we now want to use the smallest rate of the trees along the path as the rate threshold τ_{r}. Specifically, let $x^{(0)}, x^{(1)}, \ldots, x^{(k_0)}$ be the sequence of states on the path, and let $T_{i_0,l}$ be the tree on the lth transition $(x^{(l-1)}, x^{(l)})$ along the path. We then define $\tau_{\mathrm{r}} = \min_{1\leqslant l\leqslant k_0} R'(T_{i_0,l})$. With this choice for τ_{r}, we are trying to ensure those trees corresponding to the approximate most likely path to failure are generated but omit trees on less likely paths to failure.

6 Cloud-Computing Model

We now consider a model depicting a cloud-computing architecture shown in Fig. 3. There is a directed edge from component type i to component type j if $j \in \Gamma_i$, and the label on the edge is $\phi_{i,j}$. NS represents a network switch; $LB1$ and $LB2$ are two types of load balancers; $FW1$ and $FW2$ denote two types of firewalls; $HV1$ and $HV2$ signify two types of hypervisors; and $SR1$ and $SR2$ are two types of server racks. The two types mean different components in the sense that we use in the paper, i.e., they can be the same model of component but they are different types because they are in different parts of the system. This organization of components specifically shows a firewall "sandwich" as described in [12].

We set $r_{NS} = r_{LB1} = r_{LB2} = r_{FW1} = r_{FW2} = r_{HV1} = r_{HV2} = 2$ and $r_{SR1} = r_{SR2} = 3$. We assume the system is operational if at least υ_i components of each type i are up. We need at least two server racks of each type $SR1$ and $SR2$ to be up to be able to parallel process across server racks. For other component types, we require at least one component of each type of the system to be up for the system to remain operational. Thus, $\upsilon_{SR1} = \upsilon_{SR2} = 2$ and $\upsilon_{NS} = \upsilon_{LB1} = \upsilon_{LB2} = \upsilon_{FW1} = \upsilon_{FW2} = \upsilon_{HV1} = \upsilon_{HV2} = 1$. Additionally, our system operates in two environments: high demand ($e = 0$) and low demand ($e = 1$). This gives us a state space of size

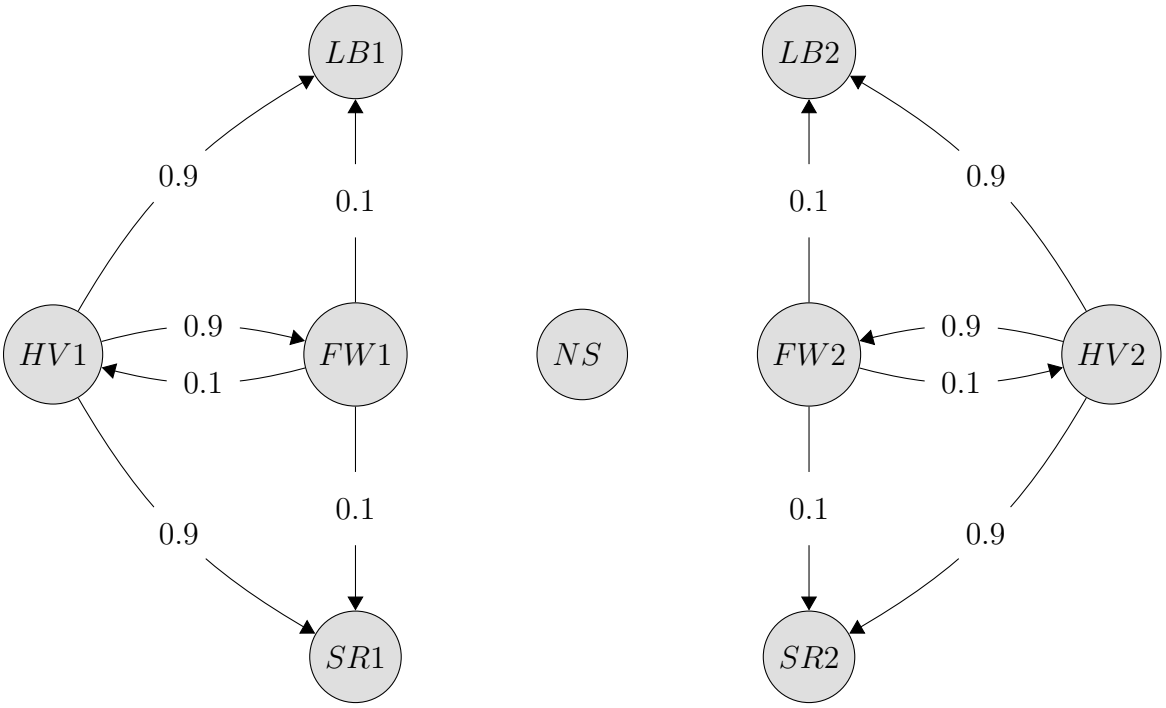

Figure 3: Cloud-computing architecture

69984. Solving this using the previous version of DECaF takes xxx seconds, whereas using the new version takes xxx seconds.

We work with a system such that all component types that are located to the left of NS (i.e., types with a "1" in their names) handle SMTP requests only, whereas all component types located to the right of NS (i.e., types with a "2" in their names) handle HTTP requests only. Thus, both groups of components (i.e., to the left and to the right of NS) need to be operational for the system to be able to handle HTTP and SMTP requests. The decoupling ensures that hardware failure on one side does not immediately propagate to the other.

As stated in [31], "hypervisors almost always cause other system components to fail and certainly cause server racks to fail because of state corruption." Thus, we assume $\phi_{HVi,LBi} = \phi_{HVi,FWi} = \phi_{HVi,SRi} = 0.9$ for $i = 1, 2$. We also assume $\phi_{FWi,LBi} = \phi_{FWi,HVi} = \phi_{FWi,SRi} = 0.1$ for $i = 1, 2$.

To estimate the component failure rates, we assume that in a high-demand (resp., low-demand) environment, a hardware component is going to fail on average in twice (resp., eight times) the time of its warranty. The time unit is hours. Typical commercial network switches, load balancers and hypervisors have warranties of 90 days (2160 hours). Thus, $\lambda_{NS,0} = \lambda_{LB1,0} = \lambda_{LB2,0} = \lambda_{HV1,0} = \lambda_{HV2,0} = 1/4320$ and $\lambda_{NS,1} = \lambda_{LB1,1} = \lambda_{LB2,1} = \lambda_{HV1,1} = \lambda_{HV2,1} = 1/17280$. Commercial server racks often have 3-year warranties giving us $\lambda_{SR1,0} = \lambda_{SR2,0} = 1/52560$ and $\lambda_{SR1,1} = \lambda_{SR2,1} = 1/210240$. Since firewalls fail due to numerous reasons such as software bugs or from targeted attacks, it is difficult to find a single representative failure rate. Thus we assume a software firewall fails on average once in five years, and set $\lambda_{FW1,0} = \lambda_{FW2,0} = \lambda_{FW2,1} = \lambda_{FW1,1} = 1/43800$.

As given in [12], failed hardware components are repaired with a rate of 1/24 regard-

less of the environment, i.e., we can replace one component a day on average. Firewalls being non-hardware components need to be rebooted after they fail, and this occurs with a rate of 10, which is also taken from [12]. The environment switches once every 12 hours on average, giving us the environment transition rates $\nu_{0,1} = \nu_{1,0} = 1/12$.

7 Concluding Remarks

8 Possible Other Stuff?

8.1 MTTF Computation

We remove the down-states from the matrix to speed up MTTF calculation.

8.2 Bookmarks

We do not store all failed components in *BFHist*, we simply multiply until we hit a not-failed component. We cache *BFHist* between succesive @ symbols to get complement cumulative component affected probabilities in constant time via array access.

Acknowledgments

This work has been supported in part by the National Science Foundation under Grants No. CMMI-0926949 and CMMI-1200065. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing, 1:11–33, 2004.

[2] C. Beounes, M. Aguera, J. Arlat, S. Bachmann, C. Bourdeau, J.-E. Doucet, K. Kanoun, J.-C. Laprie, S. Metge, J. Moreira de Souza, D. Powell, and P. Spiesser. SURF-2: A program for dependability evaluation of complex hardware and software systems. In The Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS-23) Digest of Papers, pages 668–673, 1993.

[3] S. Bernson, E. de Souza e Silva, and R. Muntz. A methodology for the specification of Markov models. In W. Stewart, editor, Numerical Solution to Markov Chains, pages 11–37, 1991.

[4] A. Blum, P. Heidelberger, S. S. Lavenberg, M. K. Nakayama, and P. Shahabuddin. Modeling and analysis of system availability using SAVE. In Proceedings of the 23rd International Symposium on Fault Tolerant Computing, pages 137–141, 1994.

[5] M. Bouissou and J. L. Bon. A new formalism that combines advantages of fault-trees and Markov models: Boolean logic driven Markov processes. Reliability Engineering & System Safety, 82:149–163, 2003.

[6] W. G. Bouricius, W. C. Carter, and P. R. Schneider. Reliability modeling techniques for self-repairing computer systems. In Proceedings of the 1969 24th ACM National Conference, pages 295–309. ACM, 1969.

[7] R. W. Butler. The SURE reliability analysis program. In AIAA Guidance, Navigation, and Control Conference, pages 198–204, 1986.

[8] B. A. Carreras, V. E. Lynch, I. Dobson, and D. E. Newman. Critical points and transitions in an electric power transmission model for cascading failure blackouts. Chaos, 12:985–1076, 2002.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. McGraw-Hill, New York, second edition, 2001.

[10] P. Crucitti, V. Latora, and M. Marchiori. Model for cascading failures in complex networks. Physical Review E, 69:045104, 2004.

[11] J. B. Dugan and K. S. Trivedi. Coverage modeling for dependability analysis of fault-tolerant systems. IEEE Transactions on Computers, 28:775–787, 1989.

[12] S. Goddard, R. Kieckhafer, and Y. Zhang. An unavailability analysis of firewall sandwich configurations. In High Assurance Systems Engineering, 2001. Sixth IEEE International Symposium on, pages 139–148, 2001.

[13] A. Goyal, P. Shahabuddin, P. Heidelberger, V. Nicola, and P. W. Glynn. A unified framework for simulating Markovian models of highly dependable systems. IEEE Transactions on Computers, C-41:36–51, 1992.

[14] C. Hirel, B. Tuffin, and K. S. Trivedi. Spnp version 6.0. Lecture Notes in Computer Science, 1786:354–357, 2000.

[15] S. M. Iyer, M. K. Nakayama, and A. V. Gerbessiotis. A Markovian dependability model with cascading failures. IEEE Transactions on Computers, 139:1238–1249, 2009.

[16] D. E. Knuth. The Art of Computer Programming: Fundamental Algorithms. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[17] G. Krishnamurthi, A. Gupta, and A. K. Somani. The HIMAP modeling environment. In Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems, pages 254–259, 1996.

[18] H. Langseth and L. Portinale. Bayesian networks in reliability. Reliability Engineering & System Safety, 92:92–108, 2007.

[19] R. G. Little. Controlling cascading failure: Understanding the vulnerabilities of interconnected infrastructures. Journal of Urban Technology, 9:109–123, 2002.

[20] H. M. Markowitz. Portfolio selection. Journal of Finance, 7:77–91, 1952.

[21] J. K. Muppala, R. M. Fricks, and K. S. Trivedi. Techniques for system dependability evaluation. In W. K. Grassmann, editor, Computational Probability, pages 445–480, The Netherlands, 2000. Kluwer.

[22] M. K. Nakayama. General conditions for bounded relative error in simulations of highly reliable Markovian systems. Advances in Applied Probability, 28:687–727, 1996.

[23] A. Peterson. oj! algorithms. http://ojalgo.org/, 2013.

[24] S. Ross. Stochastic Processes. Wiley, New York, second edition, 1995.

[25] R. A. Sahner, K. S. Trivedi, and A. Puliafito. Performance and Reliability Analysis of Computer Systems. Kluwer, Boston, 1996.

[26] K. J. Sullivan, J. B. Dugan, and D. Coppit. The Galileo fault tree analysis tool. In Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing, pages 232–235, 1999.

[27] K. S. Trivedi. Probability and Statistics with Reliability, Queueing, and Computer Science Applications. Wiley, New York, second edition, 2001.

[28] M. Walter, M. Siegle, and A. Bode. Opensesame–the simple but extensive, structured availability modeling environment. Reliability Engineering & System Safety, 93:857–873, 2008.

[29] M. Xie, Y. S. Dai, and K.L. Poh. Computing Systems Reliability: Models and Analysis. Kluwer Academic, New York, 2004.

[30] H. Xu, L. Xing, and R. Robidoux. DRBD: Dynamic reliability block diagrams for system reliability modeling. International Journal of Computers and Applications, 31, 2009. DOI: 10.2316/Journal.202.2009.2.202-2552.

[31] M. Ye and Y. Tamir. Rehype: Enabling vm survival across hypervisor failures. ACM SIGPLAN Notices, 46(7):63–74, 2011.

[32] J.-F. Zheng, Z.-Y. Gao, and X.-M. Zhao. Clustering and congestion effects on cascading failures of scale-free networks. Europhysics Letters, 79:58002, 2007.