Published in the Fifth International Workshop on Performability Modeling of Computer and Communication Systems (PMCCS-5), September 15-16, 2001, Erlangen, Germany.

Efficient verification of performability guarantees

Guillermo A. Alvarez, Mustafa Uysal, Arif Merchant
Hewlett-Packard Laboratories, 1501 Page Mill Rd., Palo Alto, CA 94304, USA
{galvarez,uysal,arif}@hpl.hp.com

Abstract

The performability of a system is usually evaluated as the mean of a performance measure, aggregated over all failure scenarios (both tolerated and not) through a Markov reward process. This metric can gloss over performance deficiencies in important portions of the failure scenario space, and is also expensive to compute because the number of possible failure scenarios can be very large. We argue that a multi-part performability specification better represents the requirements of system designers and present a fast algorithm for determining whether a given system meets such specifications.

1 Introduction

Practical computer and communication systems typically incorporate some level of fault tolerance, provided by redundant components that may fail independently. Between the occurrence of failures and the subsequent repair, the performance of the system will in general differ from that of the failure-free case (in particular, it can degrade to "no service at all" if the failures cannot be gracefully tolerated). The concept of *performability* [5] captures the combined performance and dependability characteristics of the system, *i.e.*, how well it performs in the presence of failures over a time interval.

Direct measurements of performability are only possible when a full-scale working prototype exists. Before that point, system designers must rely on *models* of the candidate configurations. Previous work on analytic performability estimation has concentrated on Markov reward models [4]: Markov dependability models (the space of all possible failure scenarios) with performance levels (the rewards) assigned to each scenario [5, 9]. Stochastic reward Petri nets provide a higher-level abstraction [1], but are ultimately solved by translating them into Markov models. Discrete-event simulators relying on fault injection are also popular, but often imply substantial development efforts and running times can be slow.

Markov reward models for realistic systems can be computationally difficult to solve because of the combinatorial explosion in the number of scenarios they contain, of numerical instability problems [8], and of the potentially high cost of computing reward levels for scenarios that contribute very little to the performability calculation. Many solutions exist to ameliorate these problems (*e.g.*, [3, 6]), where the full space is either first completely generated (if possible at all given the available resources) and then automatically pruned, or generated in condensed form by relying on domain-specific knowledge and/or human assistance to lump scenarios together. The main contribution of this paper is a novel operational technique for determining whether a candidate configuration for an arbitrary system meets performability requirements. Our algorithm generates only the indispensable portion of the failure scenario space, by first examining the scenarios with largest probabilities of occurrence and stopping as soon as a decision can be made. This is not done at the expense of accuracy; an exhaustive search would yield exactly the same results. We decouple the generation of failure scenarios from the computation of a performance level for each scenario (so they can be solved by independent models), and rely on heuristics to do as little as possible of both. Results show that our algorithm works very well for our case study of a planetary-scale content delivery network: on the average, it only took a few minutes of computation time to determine whether each sample system can or cannot satisfy a given performability requirement.

Many alternative performability metrics can be evaluated for any given system [7, 8, 9], but the most common ones fail to capture the system designer's requirements. Another contribution of this paper is a natural, multi-part way of specifying performability requirements, as a set of performance levels with associated time durations.

2 Specifying performability requirements

We identified some desirable properties of performability specifications while studying QoS guarantees in storage systems [10, 11]. First, system designers care about the fraction of the time in which *outages* occur (*i.e.*, when performance deviates from baseline goals); but it is also necessary to quantify the extent to which performance degrades during an outage, for arbitrarily low performance may not be acceptable during most of the time. Metrics that characterize the *distribution* of rewards over a period of time can capture this level of information, whereas metrics (such as expected reward) that aggregate rewards observed during both outage and baseline intervals into an average [7] cannot. Second, specifications should be efficiently and accurately verifiable against candidate configurations. In particular, for many aggregate measures it is necessary to consider all possible failure scenarios before determining whether the designer's requirements are met. Given the exponentially large size of the scenario space, this is clearly impractical and wasteful.

We therefore believe that designers should adopt *multi-part* performability specifications that reflect the individual characteristics of each performance level users are able to tolerate. A performability specification consists of $n > 0$ pairs $(r_1, f_1), (r_2, f_2), \ldots, (r_n, f_n)$, where $r_1 \succ r_2 \succ \ldots \succ r_n$ and $0 < f_1 < f_2 < \ldots < f_n < 1$. Each r_i denotes a steady-state performance level, and the corresponding f_i denotes the fraction of the system's lifetime during which its performance r must be as good as r_i or better (denoted as $r_i \preceq r$; we also use the variations $r_i \prec r$, $r_i \succ r$, $r_i \succeq r$). For example, if a web server has a guaranteed latency of at most 100ms during 60% or more of the time and at most 500ms during 95% or more of the time, the corresponding specification is (100ms,0.6),(500ms,0.95). A multi-part performability specification can be thought of as a bound on the cumulative distribution function (*CDF*) of the performance metric. Figure 1 shows a performability specification corresponding to our example and the CDFs of latency distribution for one conforming and one non-conforming configuration.

Let $S = \{S_1, \ldots, S_M\}$ be the set of all failure scenarios the system can be in. Associated with each scenario S_i is $OP(S_i)$, its probability of occurrence, and a performance level $U(S_i)$, the performance experienced in that scenario. To verify whether a performability specification is satisfied, we need to determine whether the sum of the probabilities of all scenarios in which performance is r_i or better is at least f_i. A system *meets* a performability requirement if:

$$\sum_{i=1}^{M} OP(S_i)\mathbf{1}(U(S_i) \succeq r_j) \geq f_j, \text{for all } j = 1, \ldots, n \qquad (1)$$

where the indicator function $\mathbf{1}(e)$ is equal to one if expression e is true and zero otherwise.

3 Verifying performability requirements

As stated, evaluating Equation (1) requires us to examine all M failure scenarios, where M can be a very large number. In this section, we present our approach for determining whether the equation holds while heuristically minimizing the amount of work. This is accomplished by (a) generating as few scenarios as possible, and (b) computing the performance levels for only the scenarios that are generated.

The architecture of the performability verifier consists of three modules, that operate on textual representations of workload and system descriptions [10]. In what follows, a *system configuration* is a description of the system being studied: the types of its components, the way they are configured, and their interconnections. A *failure scenario* is a system configuration plus additional information about the failures that have occurred and have not yet been repaired.

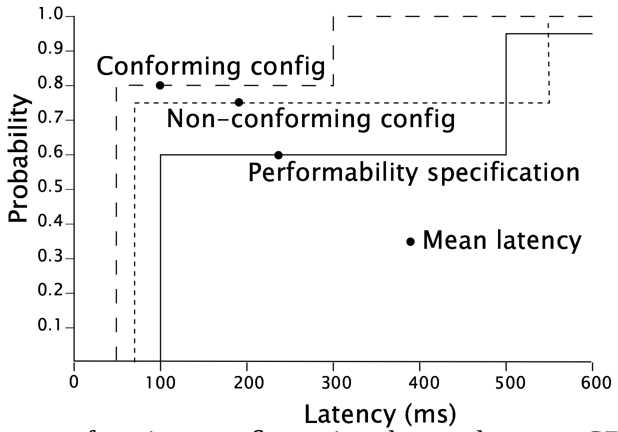
Performance predictor: Given a workload specification and a failure scenario, it returns a prediction of the performance that will be observed when servicing the input workload while in the input failure scenario. This module can be implemented in a variety of ways (*e.g.*, a simulator, or an analytical model solver).

Failure-scenario generator: This module takes as input a system configuration, and generates failure scenarios in order of decreasing probability. The ability to generate the most likely scenarios first is critical to verifying that the configuration meets performability requirements with a minimum number of performance predictions, and thus minimizing the computational requirements. If a system has components C_1, C_2, ..., C_n which fail independently of each other with corresponding availabilities a_1, a_2, ..., a_n, the occurrence probability of a failure scenario S with components C_{i1}, C_{i2}, ..., C_{ik} failed is

$$OP(S) = \left(\frac{(1-a_{i1})(1-a_{i2})\cdots(1-a_{ik})}{a_{i1}a_{i2}\cdots a_{ik}}\right)\prod_{j=1}^{n} a_j$$

In general, generating the failure scenarios in order of decreasing probability can be shown to be NP-hard, by reduction from the Subset-product problem [2]. However, the following heuristic works quite well. Suppose, without loss of generality, that $a_1 \leq a_2 \leq \cdots \leq a_n$. We generate the failure scenario with zero failures, then all scenarios with one failure, then those with two, and so on. The scenarios with k failures are generated by picking k components which are marked as failed; the remaining components are available. We generate the failed component sets in lexicographic order: $\{C_1, C_2, \ldots, C_{k-1}, C_k\}$, $\{C_1, C_2, \ldots, C_{k-1}, C_{k+1}\}$, $\{C_1, C_2, \ldots, C_{k-1}, C_{k+2}\}$ and so on.

Performability evaluator: This module operates on a system configuration, a workload description, and a performability requirement specification; it determines whether the system meets the performability specification for the input workload. The algorithm runs only as long as a decision cannot be made, and is correct for any failure-scenario generation order. This is a high-level description of the algorithm:



The conforming configuration has a latency CDF higher than the performability specification line in the entire range of latency values, while the latency CDF of the non-conforming configuration is below the bound between 500–550ms. Notice that the mean latency of the failing configuration is, at 190ms, lower than the bound prescribed by the performability specification (235ms). A verification based on this common aggregate metric would have found this configuration acceptable, even though it is non-conforming in a part of the range.

Figure 1: Sample performability specification and performance CDFs.

```
$i \leftarrow 1$
while a decision has not been made
    obtain $S_i$, $OP(S_i)$
    compute $U(S_i)$
    if $(\forall j \in [1,n])(\sum_{k=1}^{i} OP(S_k)\mathbf{1}(U(S_k) \succeq r_j) \geq f_j)$
        decide "system meets requirements"
    if $(\exists j \in [1,n])(\sum_{k=1}^{i} OP(S_k)\mathbf{1}(U(S_k) \prec r_j) \geq (1 - f_j))$
        decide "system does not meet requirements"
    $i \leftarrow i + 1$
```

The failure-scenario generator provides the next (S_i,$OP(S_i)$) pair, and the performance predictor computes $U(S_i)$. The above algorithm computes progressively tighter lower and upper bounds on the latency CDF of the given configuration; it reports a conforming configuration when the lower bound is known to be above the performability specification for the entire range, or a non-conforming configuration when the upper bound falls below the specification in any part of the range, as shown in Figure 1.

4 Case study: Wide-area content provider

We study a system in which a number of geographically-dispersed servers store and deliver *objects* to a number of user locations, also geographically dispersed. Each object served (*e.g.*, a web page) consists of several *sub-objects* (*e.g.*, advertising banners and the page's text); each sub-object is replicated on multiple different servers. We assume that network latencies between servers and users are known in advance. To retrieve an object, a user sends one request per sub-object (all in parallel) to the location that has minimum latency to the user, among the locations that have not crashed and contain a replica of that sub-object. The latency of the object's retrieval is the maximum of the latencies of retrieving the sub-objects. For simplicity, we consider only a single object and a single user location. We also assume a failure model where only servers can fail and they do so independently and in a fail-stop manner.

The content provider's network is specified by (1) a number of servers, each with given availability and latency to the user location, (2) an object, constituted of a set of sub-objects, each with a given number of replicas and their locations, and (3) a performability requirement for the latency of retrieving the object. We then apply our performability verifier to determine which configurations meet the requirements, and report the number of failure scenarios examined to complete each verification. We applied our algorithm to three such networks, using availability and latency values obtained from real-world planetary-scale networks as follows:

1. **Sites from Netcraft:** This is a 50-server network. We assigned availabilities to the servers based on the uptimes of the 50 most-available ISP sites, obtained from `netcraft.com`. We used a downtime of 4 hours/uptime to estimate their availabilities (the minimum availability was 99.89%).

2. **Sites from Uptimes:** A 27-server network with availability values obtained from `uptimes.net`. We eliminated sites which reported either 0% or 100% availabilities unless the polling period was over 500 days. Availabilities for these sites range from 17.2% to 100%.

3. **Self-reporting sites:** We used a network of 32-randomly chosen internet sites that report their own availability statistics. The minimum reported availability in this sample was 93.27%.

We assigned latency values to the servers based on average round-trip latencies of a randomly selected 100 routers spread over 5 continents, as reported by `www.internettrafficreport.com`. We assumed that every object consists of 5 sub-objects. A random, uniformly-distributed number of replicas of each sub-object are stored at randomly-chosen servers. We verify each such random configuration against three performability specifications corresponding to different service levels, from most to least stringent: *premium* ((100ms,0.96), (300ms,0.97), (400ms,0.98), (500ms, 0.99)), *standard* ((200ms,0.9), (300ms, 0.95), (500ms, 0.99)), and *economy* ((200ms,0.1), (300ms, 0.5), (500ms, 0.8)).

Our results indicate that the algorithm proposed in this paper indeed works well in practice. Table 1 presents the average number of scenarios examined by our algorithm for the case study. Running times on a single processor of an HP 9000-N4000 server were under 1 minute in all cases for the *netcraft* and the *selfreport* datasets. The *uptimes* dataset took an average of 10 minutes and a maximum of 35 minutes. The average time to examine a single scenario was approximately 5ms for our simple performance predictor; for more complex predictors, we would expect longer run times. We were able to determine whether a given performability specification is satisfied

Service levels		*Premium*			*Standard*			*Economy*		
		Mean replica count			Mean replica count			Mean replica count		
Dataset	Scenarios	5	10	15	5	10	15	5	10	15
netcraft	1.12×10^{15}	37	37	37	38	37	37	1	1	1
selfreport	4.29×10^{9}	818	1546	1567	3727	2055	519	18	17	17
uptimes	1.34×10^{8}	1226	87555	346874	177950	260193	260286	32944	48978	108632

Table 1: Number of scenarios examined by our algorithm to validate each performability specification on each of three sample content-delivery networks, for three mean replica counts. We report averages from five different executions.

by examining as little as one scenario; the maximum number of scenarios that our algorithm had to examine was 433,498, for the *uptimes* dataset. In contrast, the number of scenarios in the full Markov reward models (up to 1.12×10^{15} scenarios for the *netcraft* dataset) implies that exhaustive methods are not viable for this case study.

The heuristic embedded in the failure-scenario generator implicitly assumes that component availabilities are high, hence scenarios with fewer failures are more likely than scenarios with more failures. This is reflected in our results: the performability guarantees for configurations in which most of the sites have high availabilities can be verified by examining a relatively small number of high-probability scenarios. This is most evident for the case of the *netcraft* dataset, where our algorithm was able to verify that the *economy* requirements can be met by examining only one single scenario. This is due to the fact that the failure-free scenario in *netcraft* has an occurrence probability greater than 94.6% and its performance was sufficient to meet the requirements of the *economy* specification. In contrast, the *uptimes* dataset, which contains several sites with low availabilities, required examining a large number of scenarios as each scenario added only a small increment of probability.

The case study also demonstrated that performance specifications with either stringent requirements (*e.g.*, the *premium* specification) or weak requirements (*e.g.*, the *economy* specification) are easier to verify than those with medium-strength requirements. This is because configurations can often be shown to fail the stringent requirements very quickly, and, conversely, to pass the weak requirements quickly. Verifying medium-strength requirements often requires the algorithm to examine a larger number of scenarios.

5 Conclusions

Many traditional performability metrics measure the system's average performance during a time interval. We showed that this falls short of the needs of business- and mission-critical systems, for it potentially hides time periods during which performance degrades to unacceptably low levels. Realistic metrics need to consider the time-dependent distribution of various performance levels.

We presented a multi-part style of specifying performability requirements, and a novel algorithm that determines whether a candidate system configuration meets the requirements. Our algorithm heuristically minimizes the number of failure scenarios evaluated, thus solving the problem very rapidly and without sacrificing any accuracy. Our case study shows that this is indeed the case—we examined several network configurations using data gathered on real planetary-scale systems. All runs finished in 35 minutes or less of computation time and had very low memory requirements, as only one failure scenario is examined at a time and then discarded. Our technique scales well to large real-world systems, and compares favorably to other existing model-solving approaches.

Acknowledgements: John Wilkes proposed the multi-part specification of performability requirements and Ralph Becker-Szendy participated in the genesis of the algorithm presented here. We thank Erik Riedel and John Wilkes for comments on an earlier draft of this paper.

References

[1] G. Ciardo, A. Blakemore, P. Chimento, J. Muppala, and K. Trivedi. Automated generation and analysis of Markov reward models using stochastic reward nets. In *Linear Algebra, Markov Chains, and Queueing Models*. Springer-Verlag, 1993.

[2] M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W.H. Freeman & Co., 1979.

[3] B. Haverkort and K. Trivedi. Specification and generation of Markov reward models. In *Discrete-event Dynamic Systems: Theory and Applications 3*, pages 219–247, 1993.

[4] R. Howard. *Dynamic Probabilistic Systems*, volume 2. Wiley and Sons, 1971.

[5] J. Meyer. On evaluating the performability of degradable computing systems. *IEEE Transactions on Computers*, C-29(8):720–731, August 1980.

[6] V. Nicola. Lumping in Markov reward processes. In *Numerical Solution of Markov Chains*. Pg. 663-6, 1991.

[7] W. Sanders and J. Meyer. A unified approach for specifying measures of performance, dependability, and performability. In *Dependable Computing for Critical Applications 4*, pages 215–237. Springer-Verlag, 1991.

[8] K. Trivedi, G. Ciardo, M. Malhutra, and R. Sahner. Dependability and performability analysis. In *Performance Evaluation of Computer and Communication Systems*. Springer-Verlag, 1993.

[9] K. Trivedi, J. Muppala, S. Woolet, and B. Haverkort. Composite performance and dependability analysis. *Performance Evaluation*, 14:197–215, 1992.

[10] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *Proc. Intl. Workshop on Quality of Service*. Springer-Verlag, June 2001.

[11] J. Wilkes and R. Stata. Specifying data availabilty in multi-device file systems. *Operating Systems Review*, 25(1):56–59, 1991.