Towards Performability Modeling of Software Rejuvenation*

Sachin Garg,† Aad van Moorsel‡

† Center for Advanced Computing and Communication
Department of Electrical and Computer Engineering
Duke University, Durham, NC 27708
sgarg@ee.duke.edu

‡ Bell Laboratories
600 Mountain Ave., Murray Hill, NJ 07974
aad@bell-labs.com

Abstract

In this paper, we discuss issues in performability[1] modeling of "software rejuvenation," a form of software fault tolerance based on occasionally cleaning up the operational environment. System factors which play a key role in such a model are identified. Among these, we comment on two issues of particular interest when modeling software rejuvenation: (1) the representation of the degradation in operational environment, and, (2) the inclusion in the model of the system monitor, on which the decision to rejuvenate is based. We also survey how each of these factors have been accounted for in previous performability models and show possible directions for future work.

1 Introduction

As software continues to become larger and more complex, it is becoming the dominant source of system failures [5]. Even though all software-induced system failures have their cause in fixable design faults, the sheer complexity of modern day software along with inherent limitations in testing make it practically impossible to produce truly fault-free software.

Among various kinds of software faults, "bugs" of a particularly elusive nature have come to light. These bugs, commonly named "Heisenbugs" [5], are characterized by their non-deterministic activation, i.e., a second execution of the software, even with the same data, may not result in a failure. Transient software failures of this nature are reported in many instances in the field [1, 6, 8, 11]. The reason behind the Heisenbug's elusiveness, during testing as well as in the operational phase, is the dependence of their activation on the operational environment. (Using the terminology in [13], the operational environment includes both the process state and the process environment, i.e., the volatile and persistent state comprising of program stack, data segments and files as well as the OS environment comprising of swap space, environment variables, time etc.) Since exactly the same operational environment which led to error and failure is unlikely to be reproduced, the failure upon a second execution is avoided. This is especially true, if the environment is deliberately changed or "cleaned."

Following this reasoning, software rejuvenation has recently been proposed to avoid failures caused by Heisenbugs. In the words of [7], software rejuvenation is the "periodic preemptive rollback of continuously running applications (i.e., software) to prevent failures in the future." The implementation of this idea involves "cleaning up the in-memory data structures, respawning the processes at the initial state, logging administrative records, etc." A typical example where rejuvenation can be beneficial is when the software experiences "memory leaks," causing a continuous reduction in the amount of free memory. Rejuvenation then would consist of garbage collection, or a hardware reboot in the worst case, to reclaim memory. Since, rejuvenation typically involves an overhead, an important research issue is to determine when and how often the software should be rejuvenated. Performability modeling of software rejuvenation enables us to answer this question.

The purpose of this paper is to identify and discuss the important issues in modeling software rejuvenation. In particular, we focus on two system factors, degradation in the operational environment and monitoring. We discuss the impact these factors have on a performability model and how a model can provide feedback for better implementation of software rejuvenation, especially with respect to monitoring. Throughout, we attempt to keep the discussion as generic as possible, treating particular implementations of software rejuvenation as special cases of a generic model.

The rest of the paper is organized as follows. Section 2 elaborates on the behavior of software which employs rejuvenation. In Section 3, first, we discuss

*The material presented in this paper has been developed during Sachin Garg's summer internship at Bell Labs, Murray Hill, summer 1996.

[1] Throughout the paper we use the term performability in a generic, not measure-bound, sense.

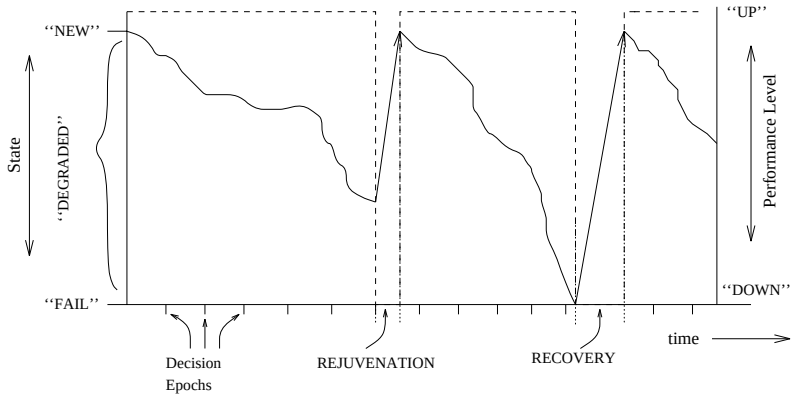

Figure 1: A realization of a stochastic process modeling software rejuvenation.

the tradeoff between white-box and black-box modeling of degradation in the operational environment. Next, we explain the inter-dependence of the performability model with the system monitor that provides data on which the rejuvenation decision is based. The 'quality' of a system monitor (that is, the ability of a monitor to distinguish where best to rejuvenate), is also discussed in this section. Finally, we survey the current literature and show possible directions for future work in Section 4.

2 Factors in Modeling Software Rejuvenation

To introduce the issues that arise when modeling software rejuvenation, Figure 1 represents the behavior of software experiencing operational environment degradation. It shows the evolution in time of a stochastic process representing the system under consideration, starting in the "NEW" state and experiencing gradual degradation, possible up to a failed situation (the state "FAIL"). The states are defined by the nature of the application, for example "FAIL" can be a complete system breakdown or a sustained poor performance. The state degradation may or may not have a direct impact on the performance. In Figure 1, for instance, two performance levels are distinguished ("UP" and "DOWN"), but degradation in state does not correspond to degradation in performance level. Of course, the extent to which the performability measure is influenced depends on the definition of this measure, which, in turn, depends heavily on the purpose of evaluation and the particular application considered. Rejuvenation is employed to prevent the failure from happening, and typically involves some cost. The technique is beneficial only when the cost due to rejuvenation is less than the cost due to failure. For instance, in Figure 1 one sees that a system failure would lead to a longer recovery time than system rejuvenation.

In an actual implementation, rejuvenation might not be possible at any given moment in time, but can only be considered at specific moments in time. These points in time are the "decision epochs," represented in Figure 1 by small vertical bars on the horizontal axis. At these epochs, a decision is taken whether to rejuvenate, based on state information obtained, for example via system monitoring. The monitored value can range from only the current time, to detailed information about memory occupancy, CPU utilization, etc.

We see that several important aspects come forward when modeling software rejuvenation, and in Section 4 we will review how previous papers dealt with issues such as the measure of interest, the kind of application and the type of rejuvenation. In a more general sense, the stochastic process in software rejuvenation models is determined by at least four aspects: it should (1) support the measure, (2) represent the system dynamic to the extend appropriate, (3) incorporate a representation of the operational environment, and (4) include the monitoring information. The latter two issues are of particular interest in software rejuvenation, and will therefore be the subject of Section 3.

3 Main Issues in Developing a Generic Model

Before discussing the modeling of the operational environment, and the consideration of the system monitor and the related decision epochs, we formalize the mathematical model under consideration. S is the state space of the stochastic process $X = \{X(t), t \in R\}$, and, for all $i \in S$, $A(i)$ is the set of actions that can be taken in state i. Let $t_0, t_1, \ldots,$ be the (real-valued) decision epochs, then a realization (sample path) of the stochastic process depends on the actions taken in the different states, at the different decision epochs. We will call the collection of actions chosen the 'strategy:' a strategy consists of the chosen action $a(i, t_j)$, for all states $i \in S$, and all decision epochs $t_j, j \geq 0$. If the performability measure [9] is $f(Y)$, where Y is a random variable, then a strategy is optimal if its actions are such that $f(Y)$ is maximized (f, a real-valued function, can for instance denote the expectation of Y).

Modeling the Operational Environment As for any model, the (base) model should have sufficient granularity to support the measure of interest, as well

as properly represent the dynamics of the system. In modeling software rejuvenation, it is important, therefore, that the degradation of the environment is captured in the failure dynamics, since it determines whether and when to rejuvenate.

When modeling the operational environment, a modeler is faced with the choice of "black-box" versus "white-box" modeling approaches, that is, between modeling the failure process, or detailing the fault or error process. Current studies have mainly dealt with a black-box approach by only modeling inter-failure times and capturing degradation via an increasing hazard rate (see Section 4). [10] uses a degrading service rate as indicator of degradation, which is very much a black-box approach since there is no modeling of the relation between environment degradation and the degrading service rate.

White-box modeling, however, still has to be pursued. It would, in the extreme case, include the modeling of faults, but, besides being unrealistic (see [12] for an illustrative discussion), for the purpose of modeling software rejuvenation a sensible white-box approach would consist of modeling the manifestation of faults as errors. Degradation, then, is modeled by modeling degradation in the components that make up the environment, such as available memory and swap space, CPU utilization, buffer queues' fill-up etc. The potential benefit of using a white-box approach to modeling the operational environment is that rejuvenation strategies are not restricted to just being "time-based". Better (and closer to optimal) strategies based on monitored and predicted behavior of individual components can be formulated.

Another issue that is relevant when modeling software fault tolerance is the effect of load on the error dynamics (see [12]). Software bugs will not show up as errors or failures if they are not activated, that is, if there is no load on the system. The degradation of the environment thus depends very much on the operational profile of the tasks that are using the system. This issue has also not been pursued explicitly in the studies we survey in Section 4.

Modeling Monitoring and Decision Epochs There exist at least two factors in an actual implementation of software rejuvenation that limit the number of strategies that can be considered: when decisions to rejuvenate can be made, and which information is available to base the decision on. A model not considering these implementation aspects can be used for the purpose of finding the optimal strategy among all possible strategies, but the identify the best strategy that is implementable, both implementation issues should be considered in the model.

Decisions to rejuvenate can only be made at the dedicated decision epochs, that is, not at just any point in time. For instance, if a decision is based on occasionally testing for awry conditions in the system, the decision epochs are the points in time that the system state is tested.

The decision to rejuvenate (at the decision epochs) will be based on the monitored system state, which is only part of the full system state (possibly including

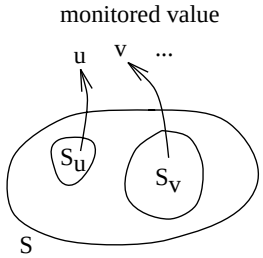

Figure 2: Different system or model states give the same monitored value.

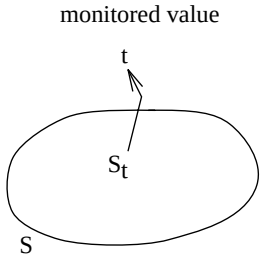

Figure 3: Only 'time' is monitored: all system or model states give the same monitored value.

time aspects). This monitoring should be represented in the model, if one wants to analyze a particular implementation. Including monitoring in the model will often change the model state space, and usually restrict the number of possible strategies. To see that the number of possible strategies decreases, consider Figure 2. Let $\mu(s)$ be the monitored value for state $s \in S$, taking values in some set U. (This discussion holds when 'state' refers to the formally defined model state as well as when it refers to an intuitive notion of system state.) Let $u \in U$, and let the set $S_u \subset S$ be composed of all states $s \in S$ for which $\mu(s) = u$. Then, for all $s \in S_u$, the *same* action must be taken, since the decision to rejuvenate is purely based on the monitored value. Hence, the choice of strategies is more restricted. The extreme but typical case that the monitor only collects one value, namely elapsed time (since the application start up), is illustrated in Figure 3.

Including implementation aspects in the model works two ways–the model better represents the system under investigation, but can only be applied to particular implementation. With regard to modeling the monitor this observation implies that the model will only be applicable if the system allows for monitoring the information on which the decision in the model are based.

The Quality of Monitors The decision to rejuvenate at a monitoring/decision epoch in an actual system implementation is based on assumptions about the future system behavior. Such assumptions conceptually resemble typical modeling assumptions. As an example, the original software rejuvenation proposal in [7] assumes that, after start-up, initially the system

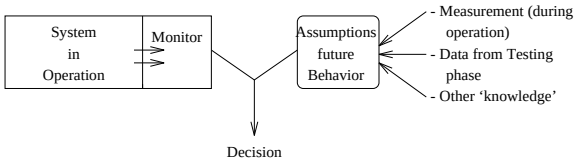

Figure 4: The decision to rejuvenate depends on monitored state plus assumptions about future system/model behavior.

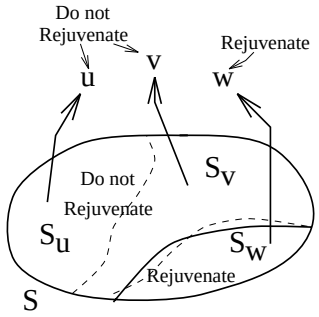

Figure 5: Decisions based on monitored value are not always optimal.

is in a state in which it will not fail, after which it gets into a failure-prone state. In Figure 4 we depict this process of decision making in a software rejuvenation implementation.

The relation between modeling and implementation suggests that the validation of the quality of the decision making is no different from general model validation. Hence, validation of the quality of decision making regarding software rejuvenation, involves checking the assumptions on which the decision is made with data obtained from system measurement. Vice versa, available data (form the testing or operational phase) gives insight into system behavior and suggests plausible assumptions about the behavior after the instant of monitoring.

Conceptually, one can determine the quality of different monitors by comparing the decision they suggest with the decision that is optimal for the considered system. As we detailed before, for any particular monitored value, there is a set of system states giving the same value. Inherent to the notion of monitoring, only one decision can be made for all the different states. Hence, a monitor has the right granularity, if all the states with the same monitored value, also give the same optimal action. We have depicted this in Figure 5. The state space S consists of a subset of states where rejuvenation is optimal, and a subset of states where rejuvenation is not recommended (the solid line in Figure 5 separates the two subsets). Assume that the monitored values are either u, v or w, where only for w rejuvenation will be carried out. A wrong decision will be made if the decision based on the monitored value does not agree with the best decision. In Figure 5, a wrong decision will be made for states in S_w outside the 'Rejuvenate' area as well as for states in S_u and S_v inside the 'Rejuvenate' area.

With regard to modeling there might be a potential area where modeling can help to validate the decision making. If the model is more detailed than the system monitor, different monitors can be compared relative to their effectiveness in determining whether to rejuvenate.

4 Previous Work and Future Directions

We end this discussion by tabulating (Table 1) the methodology adopted by previous performability models with respect to various issues discussed earlier. The second column lists how degradation was modeled in previous papers. For example, [2, 4, 7] assume that the software makes a transition from "robust" state to a state where it can "fail" in a random time. The third column shows that none of the previous studies have considered the dependence of degradation on the system load. The fifth column lists for each paper, that part of the model state which must be monitorable in an implementation (see also Section 3). After rejuvenation is initiated, the software is assumed to be completely unavailable for service and is represented in the sixth column. The optimization approach (static vs. dynamic) taken in each of the papers is listed. In "static optimization," a rejuvenation policy is assumed, therefore, as shown in column four, the decision epochs are predetermined. For example, in [2, 3, 4] rejuvenation is assumed to be performed at regular intervals of time, and the issue in model solution is to determine the optimal interval. In dynamic optimization the objective is to determine the optimal policy as opposed to tuning a particular one. Therefore, the decision epochs are unrestricted. Such an approach is followed in [10] with the use of discrete-time Markov decision models, although the solution method is based on discretization which (in a way) restricts the epochs to time points Δ apart.

Based on Table 1, we now point out possible future directions in performability modeling of software rejuvenation. First, as can be observed from the table, most of the previous studies have largely followed the black-box approach to modeling degradation. Modeling degradation of the individual components and combining to model overall system degradation is a potential approach which deserves attention. Second, as large and sudden loads have been known to be major causes of transient failures [11], the dependence of the degradation on system load could be incorporated in the performability model. None of the previous studies have done so. Third, more advanced monitors could be modeled to see whether additional information leads to significantly better performability. Finally, only a single model for rejuvenation severity has been assumed so far. In general, during rejuvenation, the software can be in any of its degraded states, an aspect that could be modeled in more detail.

5 Conclusion

In this paper we identified important issues in modeling software rejuvenation. In particular, we argued

Paper	Capturing Degradation	Load Dependency	Decision Epochs	Must be Monitorable	Rejuv. Severity	Measure	Optim. Approach	Solution Technique
[7] (FTCS95)	"failure-prone" state	not considered	predetermined	state change	unavailable	steady state availability	static	CTMC
[4] (SRE95)	"failure-prone" state	not considered	predetermined	time	unavailable	availability (point in time)	static	MRGP
[2] (FTS95)	"failure-prone" state	not considered	predetermined	time and # in queue	unavailable	# jobs lost	static	CTMC
[3] (SIGM96)	aging by non-exponential	not considered	predetermined	time and state change	unavailable	completion time	static	first moment
[10] (Perf96)	decreasing service rate	not considered	unrestricted (Δ time)	time, # in queue and service rate	unavailable	# jobs lost	dynamic	discrete-time MDP

Table 1: Modeling assumptions in previous work.

in favor of a more in-depth modeling of the degradation of the operational environment, and argued that monitoring should be included, if a particular implementation of software rejuvenation is considered. Furthermore, a survey and classification of previously published modeling studies has been given.

Both above mentioned issues are also important if on-line optimization based on monitoring and other management functionality is considered. If a system manager is used to determine when to rejuvenate, the quality of the decision making depends on the usefulness of the monitored system data, and on assumptions about future behavior which are very much similar to those made in a modeling study. In particular, we have outlined an approach to establish the quality of a system monitor. The practical utility of such quality concepts, as well as the practical feasibility of optimizing the application of software rejuvenation based on a system manager, is currently being investigated.

Acknowledgments The authors would like to thank Chandra Kintala, Yennun Huang and Kishor Trivedi for their insightful comments and useful discussions.

References

[1] A. Avritzer and F. Weyuker, "Monitoring smoothly degrading systems for increased dependability," Submitted for publication, 1995.

[2] S. Garg, Y. Huang, C. Kintala, and K. S. Trivedi, "Time and load based software rejuvenation: Policy, evaluation and optimality," in *First Conference on Fault Tolerant Systems*, Madras, India, December 1995.

[3] S. Garg, Y. Huang, C. Kintala, and K. S. Trivedi, "Minimizing completion time of a program by checkpointing and rejuvenation," in *ACM SIGMETRICS*, pp. 252–261, Philidelphia PA, May 1996.

[4] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi, "Analysis of software rejuvenation using Markov regenerative stochastic Petri nets," in *Sixth International Symposium on Software Reliability Engineering*, pp. 180–187, Toulouse, France, October 1995.

[5] J. Gray, "Why do computers stop and what can be done about it?," in *Fifth Symposium on Reliability in Distributed Software and Database Systems*, pp. 3–12, Los Angeles, January 1986.

[6] J. Gray and D. P. Siewiorek, "High-availability computer systems," *IEEE Computer Magazine*, vol. 19, pp. 3–12, January 1986.

[7] Y. Huang, C. Kintala, N. Koletis, and N. D. Fulton, "Software rejuvenation–design, implementation and analysis," in *25th Fault-tolerant Computing Symposium*, pp. 381–390, Pasadena, CA, June 1995, IEEE, IEEE Computer Society.

[8] I. Lee, *Software dependability in the operational phase*, PhD thesis, University of Illinois, Urbana-Champaign, 1995.

[9] J. F. Meyer, "On evaluating the performability of degradable computing systems," *IEEE Transactions on Computers*, vol. 29, no. 8, pp. 720–731, 1980.

[10] A. Pfening, S. Garg, M. Telek, A. Puliafito, and K. S. Trivedi, "Optimal rejuvenation for tolerating soft failures," *Performance Evaluation*, 1996. To appear.

[11] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability–a study of field failures in operating systems," in *21st Fault-Tolerant Computing Symposium*, pp. 2–9. IEEE, IEEE Computer Society, 1991.

[12] A. T. Tai, J. F. Meyer, and A. Avizienis, *Software Performability: From Concepts to Applications*, Kluwer, Norwell, MA, 1996.

[13] Y.-M. Wang, Y. Huang, P. Vo, P.-Y. Chung, and C. Kintala, "Checkpointing and its applications," in *25th Symposium on Fault Tolerant Computer Systems*, pp. 22–30, Pasadena, CA, 1995, IEEE, IEEE Computer Society.