Managing Performance of Aging Applications via Synchronized Replica Rejuvenation*

Artur Andrzejak[1], Monika Moser[1], and Luis Silva[2]

[1] Zuse Institute Berlin (ZIB)
Takustraße 7, 14195 Berlin, Germany
Email: {andrzejak, moser}@zib.de
[2] Dep. Engenharia Informática
Univ. Coimbra, Portugal
Email: luis@dei.uc.pt

Abstract. We investigate the problem of ensuring and maximizing performance guarantees for applications suffering software aging. Our focus is the optimization of the minimum and average performance of such applications in virtualized and non-virtualized scenario. The key technique is to use a set of simultaneously active application replica and to optimize their rejuvenation schedules. We derive an analytical method for maximizing the minimum "any-time" performance for certain cases and propose a heuristic method for maximization of minimum and average performance for all others. To evaluate our method we perform extensive studies on two applications: aging profiles of Apache Axis 1.3 and the aging data of the TPC-W benchmark instrumented with a memory leak injector. The results show that our approach is a practical way to ensure uninterrupted availability and optimize performance for even strongly aging applications.

1 Introduction

Problem statement. Software aging or rather *software running image aging* is the phenomenon of progressive degradation of running software image which might lead to performance reduction, hang ups or even crashes [1]. The primary causes are exhaustion of systems resources, like memory-leaks, unreleased locks, non-terminated threads, shared-memory pool latching, storage fragmentation, or data corruption. This undesirable phenomenon has been observed in enterprise clusters [2], telecommunications systems [1], web servers as well as other software. It is most likely to manifest itself in long-running or always-on applications such as web and applications servers, components of web services, and complex enterprise systems.

The primary method to fight aging is software rejuvenation, i.e. a restart of the aging application periodically or adaptively. While a lot of a research has

* This research work is carried out in part under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265) and the SELFMAN project funded by the European Commission.

been devoted recently to adaptive software rejuvenation [2,3,4,5], the remaining negative and serious side effect of a rejuvenation is the temporarily outage of service. Initiatives such as Recovery Oriented Computing (ROC) [6] and research on micro-reboots could reduce the rejuvenation time considerably. However, they require changes of the original applications and still cannot ensure uninterrupted service.

Due to their long running times service-oriented applications are especially prone to aging. We focus on this type of software and assume that invocations are triggered by external requests (we do not cover software whose invocations are triggered by timers or internal events). For such SOA-based IT-infrastructures an approach to eliminate completely the outage due to rejuvenation has been presented in [7]. The idea is to maintain in a stand-by mode an exact replica of the running application and perform an instantaneous migration in the situation when the original application is about to be rejuvenated. During the migration no requests are dropped - it is completely transparent for the users. This approach uses virtual machines as containers for the replica in order to avoid the need for additional hardware. The study [7] used one active replica at a time only and un-optimized performance thresholds as rejuvenation triggers. This causes two shortcomings: a large variation of the application performance and wasted application capacity due to non-optimal rejuvenation schedules.

Paper idea and contributions. The idea proposed in this paper is to hold multiple *active* replicas of the aging application, and trigger the rejuvenation of each one according to an *optimized schedule*. We implement this schema by simulating the system of multiple replicas and using a genetic algorithm to find such schedules. The optimization can maximize either the "any time" cumulative performance (minimum performance), the average cumulative performance (averaged over many rejuvenation phases), or a mix of both. Our approach can be used equally well in a virtualized environment on a single-server or in a cluster of native deployed applications.

Compared with the approach used in [7] our work ensures higher levels of "any time" cumulative performance, better overall utilization levels and higher resilience to unexpected performance changes or failures of individual replica. Another benefit is a smaller variation of the instantaneous performance, as with k active replicas the cumulative performance during rejuvenation is roughly $(k-1)/k$ of the maximum. Furthermore, for the case of our data running k replicas in parallel slows down the aging progress by a factor of k. As for drawbacks, our approach is more involved as it requires a rejuvenation scheduler and models of the aging processes [8].

This paper provides following contributions:

- we obtain analytically the optimized rejuvenation schedules for the minimum cumulative performance for the case of two identical replicas
- we propose and implement a heuristic method for finding optimized schedules for equal or different aging profiles of the replicas based on simulating the chains of rejuvenations and optimization via genetic algorithms

- we perform an extensive set of experiments to investigate the optimized rejuvenation schedules for a multitude of scenarios using the following data:
  - a TPC-W benchmark coupled with a fault injector to produce memory leaks (512, 768 or 1024 bytes) at each request
  - Apache Axis 1.3/1.4 server which suffers under severe "natural" aging problems
- we show that for our datasets the virtualization overhead does not depend on the number k of replicas and that using k replicas slows down the aging process by a factor of k.

Paper structure. Section 2 discusses related work. In Section 3 we introduce definitions and derive an analytical method to maximize the minimum performance of two replicas. In Section 4 we describe the idea and implementation of the heuristic optimization method. Section 5 is devoted to the experimental results, and we conclude with Section 6.

2 Related Work

The major tool to combat the problems related to software aging is software rejuvenation. There are two major approaches in this domain: *periodical rejuvenation* based on time or work performed, and *adaptive* or *proactive rejuvenation* [2,3,4,5] where the time to resource depletion or performance degradation is estimated. Countless studies have shown that the latter approach is more efficient, resulting in higher availability and lower cost.

Among the methods to apply proactive software rejuvenation two are dominant: analytic-based approach, and the measurement-based approach. The first method attempts to obtain an analytic model of a system taking into consideration various system parameters such as workload, MTTR and also distributions of failure. On this basis, an optimized rejuvenation schedule is obtained. The tools used here include continuous-time Markov chain models [9], semi-Markov models [10], and others [11].

The measurement-based approach the goal is to collect some data from the system and then quantify and validate the effect of aging in system resources [3]. The work presented in [2] considers several algorithms for prediction of resource exhaustion, mainly based on curve-fitting algorithms. Our previous work [8] used spline-based aging models to obtain optimized rejuvenation schedules. While these results are related to this work, the focus in [8] is on a single server or application.

The Recovery Oriented Computing (ROC) [6,12] project form Stanford and Berkeley focuses on minimizing the negative side effects of the rejuvenation or in general recovery phases. While the ROC-based approaches can substantially increase the up time, they require modifications of the application code.

Object and process-level migration are very well-studied techniques for providing fault-tolerance in distributed systems [13,14]. However, they add substantial cost to the software development and increase the overall system complexity.

Moreover, they do not guarantee resilience against aging, as the faulty process/object state might be migrated as well. Checkpointing-based schemes [15] suffer from similar drawbacks. In contrast, the approach discussed here does not require code modifications and can be used with legacy or black-box software.

Virtualization has proved as a successful tool for management of complex IT-environments and it is emerging as a technique to increase system reliability [16,7]. It has been exploited in [16] for proactive migration of MPI tasks from health-deteriorating to healthy hardware nodes. Work presented in [7] uses virtual machines with application replica to completely eliminate the service outage during the rejuvenation. Contrary to this work we consider a scenario of multiple simultaneously active replicas and optimize the rejuvenation schedules.

3 Maximizing Performance of Aging Applications

In the following we use the terms performance and throughput interchangeably, where latter is the number of served requests per second. We consider the scenario of an application consisting of two or more *replicas* which provide the same service. The term *cumulative* is used when all replicas are involved, otherwise we speak of an *individual* replica. The *instantaneous performance* P_X of a replica X is defined as the maximum number of requests per time unit which it can handle. An analogous performance definition is assumed for the cumulative case and is denoted as P_{cum}.

If k replicas are running simultaneously we can rejuvenate one of them without interruption of availability. During rejuvenation the instantaneous performance (throughput) decreases by about $1/k$. The choice of a proper rejuvenation schedule is critical to guarantee the cumulative performance characteristics. We are especially interested in the *minimum (cumulative) performance* P_{min} and the *average (cumulative) performance* $\bar{P}$ over longer time intervals (many rejuvenation cycles). The earlier is defined as the minimum instantaneous performance accumulated over all k replicas during the whole considered operation interval. The latter is the number of requests served cumulatively divided by the total time in which they have been served.

Due to aging effects the individual instantaneous performance is not constant and so it is represented as a function called *aging profile*. Following the study in [8] we assume that an aging profile is as a function of the number w of served requests since the last rejuvenation, i.e. $P_X = P_X(w)$.

An essential parameter is the number of requests dropped during the rejuvenation by an individual replica. We denote this number by D. Its value depends on the actual (and unknown) service rate distribution. However, it can be bound from above as the product of the rejuvenation time and the maximum instantaneous performance of a replica.

3.1 Optimizing the Minimum Performance

We consider in the following the problem of maximizing P_{min} for the case of two replicas running simultaneously. When replica A is rejuvenating, B is completely

responsible for the cumulative performance, and so the rejuvenation phase of A should be chosen during the highest performance of B. This implies that the start of $A's$ rejuvenation should be dependent on the current state of performance of B. Since the latter is determined by $P = P(w)$ and w, we introduce d_A as the number of requests served by B (since $B's$ rejuvenation) which we count until A should be rejuvenated. We call d_A the *delay* of A, and define d_B analogously for B. Since both replicas are identical, we might assume that the best solution is symmetric, and so $d_A = d_B$.

Our experience shows that the aging profiles usually consists of a *build-up phase* when the performance goes from 0 to a peak, and the *decay phase* when a performance drops monotonically from the peak until a complete crash, see Figure 1. This type of behavior is typical for aging processed caused by successive depletion of resources and widely encountered in software systems, see discussion at the end of Section 5.1. Sometimes secondary aging effects or inherent system characteristics can cause the profile to be more "random", e.g. exhibit multiple performance "jumps" before crash. Our approach does not work if this randomness is too large. To eliminate these cases, we use the aging modeling schema developed in [8] which provides a test whether the aging behavior is sufficiently "deterministic" and so our assumptions are applicable.

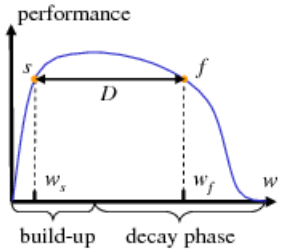

Fig. 1. Computing optimal rejuvenation schedule for the case of two identical replicas

Based on these aging properties, the idea is to schedule the rejuvenation of A such that B is performing at the "top" of its aging profile while A is not available. It is not hard to see that for reasonably small values of D there always exist two unique points $s = (w_s, P_s)$, $f = (w_f, P_f)$ on the aging profile with the following conditions, see Figure 1:

- s is in the build-up phase, and f is in the decay phase,
- their horizontal distance $w_f - w_s$ is exactly D,
- their respective performance level is the same, i.e. $P_s = P_f$.

Obviously the solution of finding P_{min} is to set the rejuvenation start of A (i.e. shut down A) such that it coincides exactly with s, i.e. $d_A = w_s$, and put A into function exactly after B served D requests. With a reasonable value of D this is always possible. Since $B's$ performance does not drop below $P_s = P_f$ during the rejuvenation, P_{min} has at least this value. Moreover, there is no segment of the aging curve of "horizontal length" D s.t. the performance inside the segment is strictly higher than P_f, and so this is also the optimum.

After this rejuvenation the roles of the replica are switched, i.e. B is rejuvenated after A has served $d_B = d_A$ requests since its restart. The points s and f can be found via a binary search on the performance (y) axis of the aging curve

with the curve peak as the upper bound. The reader might note that the value of d_A determines the length of the rejuvenation phase, and so the average number of requests. Therefore, optimization of P_{min} might conflict with the optimization of the average performance.

Figures 2 show that that the build-up phase might not exist. This is a special case of the above discussion, and here the solution is obviously to rejuvenate B right at the start of the other replica. Other types of aging profiles (especially with several local maxima) require further refinement of this approach. Intuitively, such a pair of points can be found via a "sweep" with a horizontal line from above until the intersections of the aging curve with the line form at least one segment whose endpoints fulfill the conditions analogous to those shown in Figure 1.

For $k > 2$ replicas (even identical) finding the solution is even more involved. One approach would be to perform the above "sweep" for any delay combination of the $k - 1$ replicas remaining active. Since this is not feasible, we propose a heuristic optimization described in Section 4.

4 Heuristic Optimization of Rejuvenation Schedules

In this section we describe the design of the heuristic rejuvenation scheduler and explain the policies used in our simulations. The basic idea is to use a simulation which evaluates the scheduling policy in combination with a genetic algorithm which searches for optimal policy parameters. Genetic algorithm optimization is a well-known technique which essentially performs a parallel hill climbing [17].

The major case specific part of the this optimization is the evaluation of a candidate scheduling policy by means of a simulation. It emulates the performance behavior of the full set of application replicas over a large number of rejuvenations. The simulation progresses over the number of cumulatively served requests and not over time, i.e. each step corresponds to a change caused by serving a fixed number of requests. In each step the requests are first distributed in the round-robin fashion according to the instantaneous performance of each simulated replica. Then the counters of the number of served requests are updated, and finally the new instantaneous performance levels are computed from the spline-based aging models.

After each step, the new state of the system is essentially determined by the number of requests served by each replica since each rejuvenation. Usage of the spline-represented aging profiles allows for determining the instantaneous performance levels for individual replicas and for the cumulative view. During the simulation the minimum cumulative performance and the average cumulative performance are recorded and later returned as the results.

The implementation has been done in Matlab 2006b. In the genetic optimization the maximum number of generations was 100 and the population size was set to 40. The running time of a single optimization was always below 1 minute on single core of an Intel Core Duo T2600 processor. These parameters were chosen to keep the running time low without affecting the quality of results.

4.1 Rejuvenation Policies

In this process each rejuvenation is initiated according to the current policy and its parameters. We tested two classes of policies:

- *delay based*: the least performing replica X (usually "oldest") is rejuvenated when the most recently restarted ("youngest") replica has served at least d_X requests,
- *performance based*: the least performing replica X is rejuvenated when the cumulative performance drops below a certain level Q_X.

Each policy is thus determined by its type and the vector of parameter values which are subject to optimization. We have also experimented with the variation that the parameters d_X and Q_X depend both on the replica X to be rejuvenated and the "youngest" replica Y, i.e. we have then $d_{X,Y}$ and $Q_{X,Y}$. However, the used profiles and the request distribution scheduling implied that the order of rejuvenation of the replicas remain the same, and so the pairs X, Y are uniquely determined by X or Y.

At the start of the simulation the replicas are added (or "started") subsequently. In the delay based case the second replica is added after the first has served d_1 requests, the third is started after the second has served d_2 requests etc. For the performance based policy the next replica is started after the previously started replica has served 10.000 requests (in the subsequent rejuvenations these shifts adjust according to the cumulative performance level). In the simulation we do not consider the initial phase and start recording performance levels when all replicas are up.

5 Experimental Studies

5.1 Experimental Setup

For our study we used data from two web service applications. Table 1 summarizes these datasets and their characteristics. For each case or a combination of settings we performed a run until complete crash to model aging by sending service requests with a constant rate exceeding the capacity of the server. The un-served requests have been dropped by the server and were not counted. We recorded the throughput (of served requests) as a function of time and as the number of served requests.

The first application is Apache Axis 1.3. We have conducted two sets of studies for this case. The first is used for observing the virtualization overhead (datasets $V_1, \ldots, V_4$), see Section 5.2. Here we run several replica ($k = 1, \ldots, 4$ in V_k) of the Axis server simultaneously, each in a separate virtual machine. Details on the parameters of the servers, virtual machines and the replicas can be found in Section 4.2.5 of [7]. The second set of experiments ($A1$ and $A2$) with Apache Axis 1.3 has been performed to record the consistency of the aging behavior and the aging profile of this server. These experiments used a non-virtualized scenario. Depending on the maximum number of total connections

the collected data gives rise to datasets $A1$ and $A2$, where the maximum number of connections for $A1$ was 20 and 25, whereas for $A2$ it was 50 and 100. The time needed for rejuvenation of a replica was about 10 seconds for this application. For more details see [8].

The second type of application was a Java implementation [18] of the TPC-W benchmark which created datasets $T1$, $T2$ and $T3$. This benchmark has been run with XEN virtual machines on top of Linux 2.6.16.21-0.25-smp. Since the original TPC-W implementation did not show any visible aging problem, we implemented a small fault-injector that works as a resource parasite: it consumes system resources in competition with the application [19]. The only difference between each setting was the size of the memory leak injected at every request, namely 1024 bytes ($T1$), 768 bytes ($T2$) and 512 bytes ($T3$). The rejuvenation time for the TPC-W software ranged between 12 and 15 seconds, with 13.6 seconds on average. Further information on the configuration values can be found in Section 4.1.3 of [7].

Table 1. Used datasets and their characteristics (VM = operated in a virtual machine)

Name	Application	VM	Aging	# Runs
V_1-V_4	Axis 1.3	yes	natural	5
$A1$,$A2$	Axis 1.3	no	natural	6
$T1$,$T2$,$T3$	TPC-W	yes	memory leak	5

To obtain spline-based models of aging we followed the approach presented in [8], and obtained models accurate within at most 8% tolerance. The accuracy of these models confirm that the studied aging process depend essentially on the *number* of served requests, and are independent on the request rate distribution or its burstiness [8]. While not all aging processes has this property, those caused by unreleased resources (such as memory leaks) are very likely to exhibit this behavior. This is a large class of aging processes (all processes encountered by the authors are of this type) which supports the experimental validity of the approach.

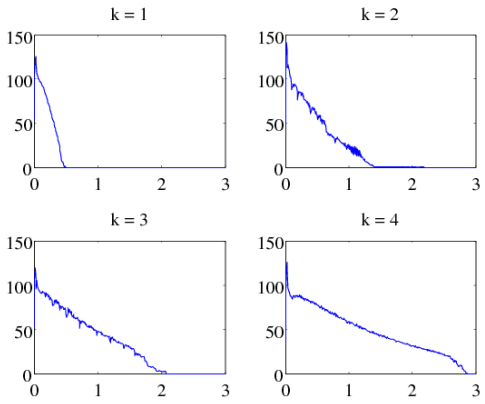

Fig. 2. Average cumulative throughput for different number of replicas as a function of time (x-axis: time in hours, y-axis: throughput in requests / second)

5.2 Virtualization Overhead and Delaying the Aging Process

According to the results in [7] running the Axis server on top of XEN virtual machine introduced a 12% overhead compared to its performance directly on Linux. We evaluate here the overhead of running different numbers of replicas in a virtualized environment by considering the throughput of k virtualized replicas of datasets $V_1, ..., V_4$ ($k = 1, ..., 4$). Table 2 implies that the overhead is not dependent on k. Simultaneously Figure 2 shows that using more replicas delays the aging process: if an application without replicas crashes after x requests, the replicated case crashes after $k\,x$ requests. This is the case as the aging behavior for the Axis application depends on the number of served requests. When using k replicas in parallel, each of them has to serve around $1/k$ requests per time and therefore it lives $k-$times longer.

Table 2. Performance measures for different number k of replicas

	# served req. (in 100s) in interval $[min, min]$						peak throughput	time to crash
	$[5, 10]$	$[5, 15]$	$[5, 30]$	$[5, 60]$	$[5, 120]$	all	$[req./sec]$	$[min]$
$k = 1$	280	510	790	790	790	790	126	30
$k = 2$	279	525	1173	1852	2026	2026	141	84
$k = 3$	275	540	1244	2299	3258	3264	119	124
$k = 4$	260	521	1261	2480	4059	4075	126	171

5.3 Schedules with Optimized Minimum and Average Performance

Table 3. Optimization results by the simulation approach using 1-parameter policies (objective: ave = average performance, min = minimum performance)

Cases	$A1 + A1$		$T1 + T1$		$T3 + T3$		$A1 + A1 + A1$		$T1 + T1 + T1$		$A1 + A1$	
Policy	delay										perf.	
Objective	ave	min	ave	min	ave	min	ave	min	ave	min	ave	min
P_{min}	422	433	55	56	56	57	833	839	111	112	302	297
P_{ave}	760	746	110	109	110	57	1138	1122	165	164	682	710
d or Q	29771	20260	35688	14139	17813	0	18946	12726	25682	9810	600	610

In this section we present the results of the optimization. We distinguish between policies with one parameter and policies with different parameters. Policies with one parameter represent the most common case, as normally all application replicas should have the same aging profile.

We have first performed the optimization of the minimal performance by the analytical approach from Section 3. The values of d are as follows: $A1 + A1$: 21600, $A2+A2$: 197500, $T1+T1$: 13000, $T2+T2$ and $T3+T3$: both 0. They agree

within a reasonable error with the results obtained from the heuristic approach for policies with one parameter (Table 3 - case $T2 + T2$ is not included there). This verifies that the simulation is correct. The latter table does not include cases with the performance based scheduling policy as this approach turned out to be more sensitive to parameter variations (see discussion below).

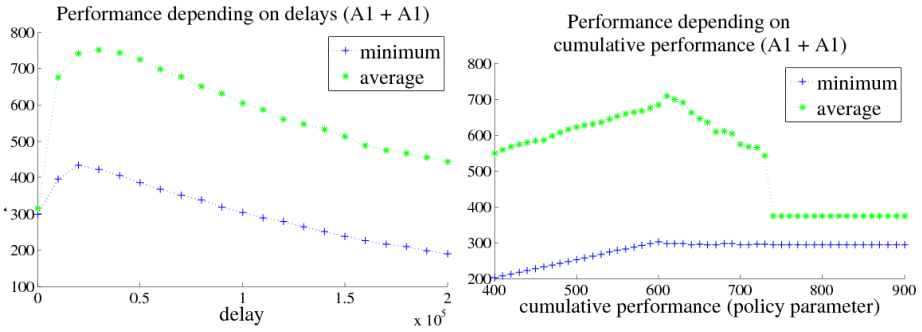

Fig. 3. Average and minimum performance plots for dataset A1 with two replicas. Left: delay based policy, right: performance based policy

Table 4 contains the results from the heuristic approach for policies with different parameters. It shows that an optimization for the average or minimum performance results in different delays.

The heuristic optimization and simulation are illustrated by two different kind of plots: simulation plots and range plots. Simulation plots show the performance history of the individual replica and the cumulative performance history depending on the number of processed requests. Contrary to this, range plots show the minimum and average performance depending on the parameter value of the rejuvenation policy.

Figure 3 shows a range plot with the performance of two identical replicas depending on either a delay based or a performance based scheduling policy. The peaks of the curves show that the optimized policy parameters are different for

Table 4. Optimization results by the simulation approach using 2 or 3-parameter policies

Cases	$A1 + A2$		$T1 + T3$		$T2 + T3$		$T1 + T2 + T3$	
Objective	ave	min	ave	min	ave	min	ave	min
P_{min}	420	431	54	56	54	56	108	110
P_{ave}	761	752	78	106	109	59	164	137
d_1	38552	22819	45111	0	51438	0	44357	0
d_2	21434	22334	7337	12733	14243	1	28157	1
d_3	-	-	-	-	-	-	823	1

the optimized minimum performance and the optimized average performance. An optimization for both performance measures at the same time is not possible in this case. The results of the range plot with a performance based scheduling policy show that choosing the right value for rejuvenation is more important than with the delay based policy. If the chosen value is too high the performance of the replicas drops fast. Also this policy is less resilient to variations in the system than a delay based policy.

As an example we show a simulation plot for a combination of different replicas (Figure 4). This is a simulation run with a delay based policy. The delays are those which were gained from the optimization described in Section 4. In settings with identical replicas, the delays have the same values. The plots confirm the results from the range plots. Optimization for the minimum performance lead to different delays than an optimization for the average performance.

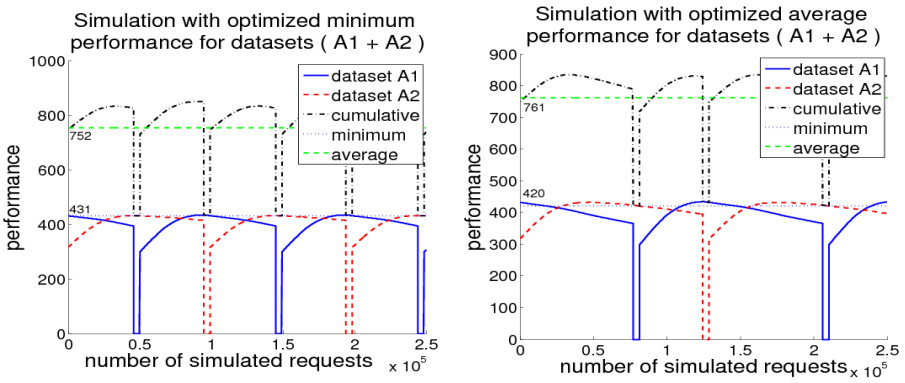

Fig. 4. Simulated performance of individual replicas (lower part) and cumulative performance (upper part). Left: delays optimized for minimum performance, right: delays optimized for average performance (each with two different replica $A1 + A2$)

6 Conclusions

Our results show that optimization of the rejuvenation schedules of simultaneously active application replicas is a practical and effective approach to combat software aging without sacrificing availability and performance. Since this approach does not require software changes, it offers a simple and non-intrusive way to reducing management costs of aging application in SOA-based environments.

Future work will include experiments with an implementation under real-world conditions to verify the practical efficacy of the approach. Furthermore, we plan to extend the approach to non-deterministic aging profiles and transient failures.

References

1. Avritzer, A., Weyuker, E.: Monitoring smothly degrading systems for increased dependability. Empirical Software Engineering **2**(1) (1997) 59–77
2. Castelli, V., Harper, R., Heidelberg, P., Hunter, S., Trivedi, K., Vaidyanathan, K., Zeggert, W.: Proactive management of software aging. IBM Journal Research & Development **45** (2001)
3. Garg, S., van Moorsel, A., Vaidyanathan, K., Trivedi, K.: A methodology for detection and estimation of software aging. In: 9th International Symposium on Software Reliability Engineering. (1998) 282–292
4. Vaidyanathan, K., Trivedi, K.S.: A measurement-based model for estimation of resource exhaustion in operational software systems. In: 10th IEEE International Symposium on Software Reliability Engineering. (1999) 84–93
5. Vaidyanathan, K., Trivedi, K.S.: A comprehensive model for software rejuvenation. IEEE Trans. Dependanble and Secure Computing **2** (2005) 1–14
6. Brown, A.B., Patterson, D.A.: Embracing failure: A case for recovery-oriented computing. In: High Performance Transaction Processing Symposium. (2001)
7. Silva, L.M., Alonso, J., Silva, P., Torres, J., Andrzejak, A.: Using virtualization to improve software rejuvenation. In: IEEE International Symposium on Network Computing and Applications. (2007)
8. Andrzejak, A., Silva, L.: Deterministic models of software aging and optimal rejuvenation schedules. In: 10th IFIP/IEEE Symposium on Integrated Management. (2007)
9. Huang, Y., Kintala, C., Kolettis, N., Fulton, N.: Software rejuvenation: Analysis, module and applications. In: FTCS-25. (1995)
10. Dohi, T., Goseva-Popstojanova, K., Trivedi, K.S.: Statistical non-parametric algorithms to estimate the optimal software rejuvenation schedule. In: Pacific Rim International Symp. Dependable Computing. (2000) 77–84
11. Garg, S., Puliafito, A., Telek, M., Trivedi, K.S.: Analysis of preventive maintenance in transactions based software systems. IEEE Transactions on Computers **47** (1998) 96–107
12. Candea, G., Kiciman, E., Zhang, S., Fox., A.: Jagr: An autonomous self-recovering application server. In: 5th Int Workshop on Active Middleware Services. (2003)
13. Chakravorty, S., Mendes, C.L., Kalé, L.V.: Proactive fault tolerance in MPI applications via task migration. In: 13th HiPC. (2006)
14. Douglis, F., Ousterhout, J.K.: Transparent process migration: Design alternatives and the sprite implementation. Software—Practice and Experience **21** (1991) 757–785
15. Stellner, G.: Cocheck: Checkpointing and process migration for MPI. In: 10th IPPS'96. (1996) 526–531
16. Nagarajan, A., Mueller, F., Engelmann, C., Scott, S.: Proactive fault tolerance for HPC with xen virtualization. In: ICS07. (2007)
17. Man, K.F., Tang, K.S., Kwong, S.: Genetic Algorithms: Concepts and Designs. Springer (1999)
18. Manjhi, A.: TPC-W in Java on Tomcat and MySQL. Carnegie Mellon University. (2005)
19. Gross, K., Bhardwai, V., Bickford, R.: Proactive detection of software aging mechanisms in performance critical computers. In: 27th Anual IEEE/NASA Software Engineering Symposium. (2002)