Software Life-extension: A New Countermeasure to Software Aging

Fumio Machida, Jianwen Xiang, Kumiko Tadano, Yoshiharu Maeno
NEC Knowledge Discovery Research Laboratories
Kawasaki, Japan
{f-machida@ab, j-xiang@ah, k-tadano@bq, y-maeno@aj}.jp.nec.com

Abstract—This paper presents software life-extension, a new technique for counteracting software aging by preventive operation to extend the lifetime of software execution. Software aging is a phenomenon of progressive degradation of execution environment due to aging-related software faults and it might cause resource depletion resulting in system failures. To extend the lifetime of the software affected by aging, we use a virtual machine to execute the software and allocate additional memory to the virtual machine upon software aging detection. Although software life-extension is a temporal solution as it only postpones the occurrence of a failure, it provides a simple, cost-effective, and non-intrusive countermeasure to software aging. The feasibility and effectiveness of software life-extension are studied by the experiments on memcached, a widely adopted general-purpose in-memory cache server. From the experimental results, we present a Semi-Markov process (SMP) describing the general behavior of software life-extension and analyze the model which gives the prediction of the system availability as well as the user-perceived availability.

Keywords-component; availability, software aging; life-extension; software rejuvenation, memcached

I. INTRODUCTION

Faults in software, often called software bugs, are one of the major causes of failures in IT systems. It is unrealistic to remove all the software bugs in the development and testing phases and hence residual bugs tend to remain in operational phase of the software. Such software bugs are found in the operations of IT systems and they sometimes induce system failures. To mitigate the adverse effects of the residual bugs, operational techniques such as availability monitoring, automatic recovery, preventive maintenance, check-pointing and backup are essential for high-available IT systems.

Aging-related bug is a type of software bug which accumulates errors in its execution environment and causes system failures and/or performance degradations. The phenomenon of progressive degradation of software execution environment which increases the failure rate of the software is called software aging [1]. A typical example of software aging is progressive increase in memory consumption which conclusively causes a memory leak. Since software aging can be observed only in the software execution, it is difficult to find aging-related bugs until the software is deployed and executed in a specific environment.

For software suffering from aging, software rejuvenation is known to be an effective countermeasure [2]. Software rejuvenation is a preventive maintenance technique which cleans up the internal states of software execution environment by restart or reset before the software faces serious performance degradation and/or failures. System failures caused by software aging can be prevented using software rejuvenation, although it involves another type of downtime overhead due to restart. On top of that, the downtime coming from software rejuvenation may cause in some applications serious damages to the operations. For example, in an application holding cache content in volatile memory, software rejuvenation clears all the memory content and the performance of the application may be degraded after restart. It is vital for such application to keep the memory content as long as possible during its mission period. Software rejuvenation is not a suitable solution in such cases.

In this paper, we present an alternative countermeasure to software aging, software life-extension, a preventive maintenance technique which prolongs the lifetime of software execution for as long as possible in face of software aging. If failures caused by software aging can be postponed for a while, users or applications may effectively use the extended residual lifetime. As an example, if the content of an application with memory cache is preserved during the extended lifetime, it might be possible to finish the user sessions within the lifetime or to save the cached content to a persistent storage before a failure. This approach is particularly useful for long running mission critical applications which require continuous up-time.

There are at least two possible approaches to implement software life-extension, namely dynamic resource allocation and workload control. The first approach extends the software lifetime by replenishing resources dynamically when the resource used by the software becomes deficient due to software aging. Such flexible resource management is particularly useful in a server virtualized environment where computing resources are virtualized and can be shared by multiple virtual machines. On the contrary, the second approach impedes the progress of software aging by reducing the workloads on the aged software. This approach is appropriate when the progress of software aging is closely associated with the amount of workload.

In this paper, first we present the concept of software life-extension. Then, the feasibility and effectiveness of software life-extension are studied by means of experiments
on memcached\(^1\), an in-memory key-value store for caching object. Memcached is widely adopted in a number of large scale web systems including Facebook\(^2\), Wikipedia\(^3\), YouTube\(^4\) and Twitter\(^5\) etc. While such memory-oriented storage systems are getting more popular, the impacts of software failures caused by software aging become significant. Software rejuvenation, which clears the whole memory content by restart, is not a suitable solution to the aging in cache servers. We examine the possibility of software life-extension using virtual machine (VM) where additional memory is allocated to the VM dynamically after aging detection. We observe that an allocation of a small portion of additional memory in the right instant of time prolongs the lifetime of memcached greatly. We also conduct accelerated life time experiments of memcached in our test bed to analyze the system availability and performance which are improved by software life-extension. The measured data is later used to construct a Semi-Markov process (SMP) which captures general behavior of software life-extension and allows us to estimate the steady-state system availability and the expected number of request drops during the downtime. The estimated values are verified with the real observations. Additionally, the performance of memcached is characterized by cache hit rate. The advantage of software life-extension is presented also in terms of cache hit rate.

The rest of the paper is organized as follows. First, Section II introduces the general concept of software life-extension and shows a simple Markov chain representing the behavior of software life-extension. Section III studies the feasibility of software life-extension by experiments on memcached. Section IV shows the results of accelerated life tests for analyzing the availability of the system using software life-extension. We model the system using an SMP to estimate the steady-state system availability and the expected number of request drops. Section V provides some discussions. Section VI describes related work and finally, Section VII gives our conclusions.

## II. SOFTWARE LIFE-EXTENSION

This section introduces the concept and methods of software life-extension. In order to explain the conceptual difference between software life-extension and software rejuvenation, an availability model representing the behavior of software life-extension is described.

### A. Concept

The term of software life-extension is an extension of the metaphor for software aging. Software aging represents the state when a software execution environment has been degrading in terms of available resources since the beginning and the lifetime of software terminates when the system fails due to resource depletion. Software life-extension is an operation extending the lifetime by impeding the progress of software aging. Although resource depletion is inevitable without fixing the aging related bug, the time to resource depletion can be prolonged by controlling the environmental factors such as resources and workloads. If we allocate additional resources to the software execution environment and the software makes use of the additional resources effectively, the time to resource depletion is prolonged according to the amount of added resources. Alternatively, in the case of software aging depending on the amount of workload, if we regulate the workload of the software to decrease, the rate of aging progress may be lowered. These approaches do not require any changes in the software source code and are often easily applicable by means of common maintenance operations, commands, or scripts. The bug can be removed by hot fix in operational phase as well \(^3\). However, our focus in this paper is on maintenance approach to mitigate software aging which does not require any source code modification.

### B. Means

There are at least two conceptual ways to implement software life-extension in software systems: dynamic resource allocation and workload control. The first approach extends the lifetime of aged software by dynamic resource allocation in which the amount of resources in face of aging is increased dynamically to some extent during the software execution. Recent advances in virtualization technologies make such dynamic resource allocation possible. For example, Xen hypervisor\(^6\) provides the functionality to virtualize hardware resources and allocate the virtualized resources to a VM. The amount of resource allocation can be changed dynamically according to system maintenance policies \([4]\). In this approach, we need standby resources which can be allocated dynamically and may be shared by other software execution environments. The use of standby resources may incur some additional costs, such as increase of resource usage cost imposed by the cloud and/or hosting services, and increase of unavailability of other services sharing the standby resource.

The second approach controls workload to decrease the loads on the aged software. This approach is limited to the applications which work with workload manager or have a load balancer in the front-end. The workload on aged software is reduced by assigning jobs to other instances or dropping job requests at the workload manager or the load balancer. Software aging is often associated with the workload processed on the software \([5][6]\), therefore the progress of aging can be impeded by workload reduction. Although it helps to extend the lifetime of the software, resource exhaustion is inevitable as long as the software continues the execution under workload. This approach can be considered as a type of degraded system that is designed to survive even in the case of system component failure. Unlike typical degradable systems, software life-extension using workload control does not guarantee the endurance of

\(^{1}\) http://memcached.org/
\(^{2}\) http://www.facebook.com/
\(^{3}\) http://www.wikipedia.org/
\(^{4}\) http://www.youtube.com/
\(^{5}\) https://twitter.com/
\(^{6}\) http://xen.org/
the software. Even after the life-extension, the software may eventually encounter a failure due to resource exhaustion. Similar to the first approach, workload control may also incur additional cost, such as overload of other instances caused by workload reallocation and request rejections at workload manager. In this paper, we focus on the first approach and study the feasibility and effectiveness of software life-extension using VM.

C. Advantages and drawbacks

Regardless of the means, the primary advantage of software life-extension compared to software rejuvenation is continuous execution of software even during software aging. Although software rejuvenation clears the aging states with relatively small downtime, it interrupts the software execution and valuable data in memory is lost. On the contrary, software life-extension can keep the software availability without any interruptions as long as possible. When applications have long running jobs and their completions are important, life-extension is preferable to rejuvenation in terms of job completion probability. Software life-extension is also suitable for applications requiring predetermined mission time. We can apply software life-extension to meet the mission time requirement when the software execution is likely to be ended before the mission time.

Another benefit of software life-extension is the capability of preservation of memory content as mentioned in the introduction. In some types of software, the accumulated data in memory is important for its service and performance. Software rejuvenation cleans such data completely, and thus it may cause a serious degradation of service quality after the rejuvenation. A typical example of such memory content is paging data in operating system. The deletion of paging data by reboot causes performance degradation as reported in [7]. In contrast, software life-extension attempts to preserve the memory content as long as possible. While the content of memory are eventually lost at the end of the life, we may wisely use the residual lifetime to take a backup or a snapshot to save the memory content to persistent storage.

As discussed earlier, software life-extension involves additional costs such as resource usage cost, degradation of the performance, and availability degradations of other services. Additional lifetime is provided by the trade-off to those costs. Compared to hot-fix approach that corrects the source code by removing the source of software aging, software life-extension does not remove the source of the aging problem. Software aging progresses even after software life-extension is performed.

D. Availability model

To characterize our approach in comparison to software rejuvenation, we introduce a continuous time Markov chain (CTMC) which represents the general behavior of software aging and software life-extension. First, Figure 1(a) shows a CTMC representing behavior of software aging as studied in [2]. Software starts the execution from UP state that is a highly robust state. After a certain time interval, the software proceeds to failure-probable (FP) state. Following that, software goes from FP state to failure (F) state which is represented by a shaded circle. This two-step failure behavior is commonly adopted in availability modeling and analysis of software aging [2][8]. After recovery from a failure, the software returns to UP state from F state. Let us denote \( \lambda_1, \lambda_2, \mu \) as the rates attached to the transitions from UP to FP, FP to F, and F to UP, respectively. By solving the CTMC, the availability of the software \( A_N \) is computed as the sum of steady-state probabilities in UP state and FP state:

\[
A_N = \pi_{UP} + \pi_{FP} = \frac{\lambda_1 + \lambda_2}{\lambda_1 + \lambda_2} \cdot \mu + \lambda_1 \lambda_2.
\]

Next, the CTMC shown in Figure 1(b) adds the behavior of software rejuvenation to the aging model. Software rejuvenation is performed when the state of the software is changed from FP state to RJ state which represents the rejuvenation state and is denoted by a shaded circle. When the rejuvenation completes, the software returns to UP state. Let us denote \( \delta_r \) and \( r \) as the rates attached to the transitions from FP to RJ and RJ to UP, respectively. The availability of the software with software rejuvenation is computed by

\[
A_R = \frac{r\mu \cdot (\lambda_1 + \lambda_2 + \delta_r)}{r\mu \cdot (\lambda_1 + \lambda_2 + \delta_r) + \lambda_1 \cdot (\lambda_2 \lambda + \mu \delta_r)}.
\]

Then we introduce a CTMC for software life-extension as shown in Figure 1(c). Instead of the rejuvenation state, we add a life-prolonging state, LP, to the aging model. The software in FP state may go to LP state by software life-extension before encountering a failure. The software will eventually fail and the state is changed from LP to F. Let us denote \( \delta_0 \) and \( \lambda_3 \) as the rates attached to the transitions from FP to LP and LP to F, respectively. Since LP state is also a working state, the availability of the software \( A_L \) is computed as the sum of steady-state probabilities in UP state, FP state and LP state:
the expression shows that the effects of software life-extension on the availability depend on the failure rates ($\lambda_1$, $\lambda_2$, and $\lambda_3$), the failure recovery rate ($\mu$), and the life-extension trigger rate ($\delta_1$). The difference between $A_N$ and $A_L$ is computed as

$$A_L - A_N = \frac{\lambda_1 \mu \delta_1 (\lambda_1 + \lambda_2 + \delta_1) + \lambda_1 \mu \delta_1}{\mu \lambda_3 \cdot (\lambda_1 + \lambda_2 + \delta_1) + \lambda_1 \lambda_3 \cdot (\lambda_2 + \delta_1) + \lambda_2 \mu \delta_1}.$$

It indicates that the sign of $A_L - A_N$ depends on the sign of the term $(\lambda_2 - \lambda_3)$. If $\lambda_3$ is smaller than $\lambda_2$, $A_L$ becomes larger than $A_N$ resulting in software life-extension being effective in terms of steady-state availability. Software life-extension should be effective when it decreases the failure rate ($\lambda_3 < \lambda_2$), therefore the derived condition looks reasonable.

To figure out the condition where software life-extension offers a better solution than software rejuvenation, we take the difference between $A_R$ and $A_L$.

$$A_L - A_R = \frac{\lambda_1 \mu (\lambda_1 + \lambda_2 + \delta_1 - r \delta_2) \cdot (\lambda_2 - \lambda_3)}{[\mu \lambda_3 (\lambda_1 + \lambda_2 + \delta_1) + \lambda_1 \lambda_3 (\lambda_2 + \delta_1) + \lambda_2 \mu \delta_1] [\mu \lambda_3 (\lambda_1 + \lambda_2 + \delta_1) + \lambda_1 \lambda_3 (\lambda_2 + \delta_1) + \lambda_2 \mu \delta_1]}.$$

Comparing the difference with 0, from the numerator, we get the inequality

$$r \lambda_3 < \frac{\lambda_1 \mu \delta_1 \delta_2}{\lambda_1 \delta_1 + \lambda_2 \delta_2 + \delta_1 \delta_2},$$

representing the condition where software life-extension achieves higher availability than software rejuvenation (i.e., $A_L > A_N$). If the value of $\lambda_3$ in the left-hand term is small enough, the above condition is likely to hold. On the other hand, if the value of $r$ in the same term is relatively large, the condition is likely not to hold which results in $A_R$ becoming larger than $A_L$.

### III. Experimental Results

This section provides an illustrative example of software life-extension on memcached. A memcached server is deployed on a VM and its lifetime is extended by software life-extension using dynamic memory allocation supported by Xen hypervisor.

#### A. Memcached

Memcached is an in-memory key-value store for caching objects. In large-scale web systems, memcached is widely used as a cache server for database to speed up query response. It simply implements a hash table whose content is read or inserted by corresponding keys. In spite of its simple architecture, memcached is scalable to multiple servers and achieves high-performance. Unlike the persistent database server such as MySQL, storage in memory is the heart of memcached which enables faster access to the data.

#### B. Problem

Memcached consumes memory for storing key-value pairs in a hash-table until it reaches the maximum limit. When the table becomes full, a subsequent insert request purges older data in least recently used (LRU) order. The limit is set to 64MB by default but it is configurable by a command option “-m”. A possible problem comes from a misreading of this limit setting. This option is used to set the maximum size of memory for cache data and it does not count the usage of memory by memcached process and associated metadata. Consequently, the actual usage of memory may be increased beyond the set maximum value. If the maximum value is not set properly within the range of available free memory in consideration with additional memory consumed by memcached process, it might cause a number of swaps and further induce a crash due to out-of-memory. Users are responsible for setting the maximum limit correctly in accordance with the available resources on the execution environment, but sometimes it is hard to estimate the available memory in advance. Human configuration errors may also occur, especially in virtualized systems.

As an illustrative example of memory leak problems in memcached, let us consider a memcached deployment on a VM. A VM is configured to 1024MB of maximum memory allocation and 512MB of initial memory allocation. The VM starts with only 512MB of memory but it can be increased up to 1024MB later. Let us suppose that a memcached is run on a VM with the “-m” option and the maximum limit is set to 1024MB by user. There is no problem in using memcached as long as the cached content does not exceed the value of 512MB that is the initial memory allocation. However, if the data inserted exceeds the value of 512MB and there is no action to increase memory allocation, memcached uses up swap spaces and the VM crashes after all due to out-of-memory.

It is not appropriate to categorize the issue as a software failure caused by aging-related bug, but the software aging can occur due to configuration mistakes by users and it will cause the accumulation of memory usage.

#### C. Software life-extension for memcached

A simple solution to the illustrative problem is dynamic additional memory allocation to the VM. If we can increase the free memory of the VM dynamically and memcached makes use of them, the life of memcached can be extended. We examined this on Xen hypervisor version 4.1. Figure 2 shows our test bed configuration. A VM is created on Xen box and 512MB of memory is initially allocated to the VM. The VM has 512MB of swap space. A single instance of
memcached is deployed on the VM and it starts with the “-m” option and set the maximum limit to 900MB. A client program generates requests to the memcached for load test that repeatedly inserts 1MB of unique data and reads it in a subsequent access. During the load generation, we do not insert any delays between consecutive requests. According to the number of insert requests, the memory consumption of memcached increases gradually because the data is cached in memory. When the number of insert events exceeds 500, memory swapping is started. If no counteractions are performed, the VM is finally crashed due to out-of-memory. The changes of free memory and swap usage in the VM during this experiment are shown in Figure 3. The VM is crashed when it use up both of the free memory and available swap spaces.

We can postpone the failure by allocating additional memory to the VM, namely by software life-extension. When the swap usage exceeds 400MB, we allocate 88MB of additional memory to the VM through the Xen’s command line interface. The total memory is increased to 600MB and the life of the VM is extended. The time to failures (TTF) observed in three experiments under the same load test are shown in TABLE I. The software life-extension prolongs the lifetime of the VM more than ten times longer than the original lifetime. Although the amount of added memory is relatively small, the lifetime is greatly improved.

The amount of free memory and swap usage are changed by software life-extension as shown in Figure 4. The steep increase in the swap usage stops at the time when software life-extension is performed but it increases again in a short period of time after using out the added free memory. The swap usage reaches 500MB around 135 seconds. The VM does not immediately fall into out-of-memory but the swap usage fluctuates around 500MB with a subtle upward trend. Figure 5 shows the observed swap usages from the time of 135 seconds to the time when the VM fails. The upward trend is confirmed by the normal approximation test using Mann-Kendall statistic [9] with 99% confidence level and Sen’s slope estimate [10] gives the estimated trend as 13.816 KB/s.

When we allocate 600MB of RAM at the beginning rather than adding later during the runtime, such life-extension effects are not always demonstrated although the VM sometimes survives more than 2000 seconds. The preliminary result implies that the combination of the amount of the initial memory allocation and the timing to allocate the additional memory is a key to extend the lifetime. We further investigate the lifetimes with different memory configurations in the following sections.

### D. Observations

Let us represent $M_{\text{max}}$ as the maximum limit (MB) set by the “-m” option of memcached. The amount of available resources in the VM is the sum of the initially allocated memory $M_{\text{RAM}}$, dynamically allocated memory $\delta_{\text{RAM}}$ and the swap space $M_{\text{swap}}$. When $M_{\text{RAM-max}}$ denotes the maximum size of the VM, the sum of $M_{\text{RAM}}$ and $\delta_{\text{RAM}}$ remains within the bound of $M_{\text{RAM-max}}$

$$ M_{\text{RAM}} + \delta_{\text{RAM}} \leq M_{\text{RAM-max}} $$

Since memcached consumes the memory up to $M_{\text{max}}$ for data cache, the VM is tolerant to out-of-memory problem as long as the following condition holds

$$ M_{\text{max}} + \alpha + \beta < M_{\text{RAM}} + M_{\text{swap}} + \delta_{\text{RAM}} $$

where $\alpha$ is the amount of memory consumption by memcached process except for the data area and $\beta$ is the amount of memory consumption by other processes on the
VM. The values of α and β are varied during the lifetime while $M_{RAM}$, $M_{swap}$ and $M_{max}$ are fixed values. We can control the lifetime of the VM by determining the value of $\delta_{RAM}$.

TABLE II shows the observed TTFs in seconds with the different combinations of $M_{RAM}$ and $M_{max}$ where $M_{swap}$ is fixed to 512MB and no software life-extension ($\delta_{RAM} = 0$). With all the values of $M_{max}$, TTF becomes longer as $M_{RAM}$ increases. When the sum of $M_{RAM}$ and $M_{swap}$ (512MB) rises above a certain threshold (around $M_{max}$=200MB), any failures of the VM are not observed in the hours of load tests. For instance, no failure is observed in the combination of $[M_{RAM}=640MB, M_{max}=900MB]$. There are no request errors in client side in this case. However, as denoted by asterisk in TABLE II, in the combinations of $[M_{RAM}=512MB, M_{max}=800MB]$ and $[M_{RAM}=384MB, M_{max}=700MB]$, some request errors are observed in the client side. These errors may be caused by busy states of memcached suffering from swapping. These configurations might encounter failures after long time execution, although they survived during the hours of tests.

Next we fix $M_{RAM}$ and $M_{max}$ to 512MB and 900MB, respectively, and apply software life-extension by using various values of $\delta_{RAM}$. The observed TTFs are shown in TABLE III. In the first set of experiments, software life-extension is performed when the size of swap usage increases over 500MB. Software life-extension does not affect the TTF until $\delta_{RAM}$ reaches 64MB but when over 64MB, it prolongs the TTF according to the amount of memory allocation.

The results are varied if the timing of software life-extension is changed. TABLE IV shows the results of TTFs where software life-extension is performed when the size of swap usage rises above 300MB. Software life-extension does not increase the lifetime in this case. The difference is most probably caused by the behavior of memory management function in operating system which is dynamically changed especially when the system suffers from the lack of resources.

From this observation, the timing of software life-extension should be set to just before resource depletion, at least in our test bed.

IV. AVAILABILITY EXPERIMENTS

In order to analyze the impacts of software life-extension on system availability and performance, we conduct further experiments on memcached in our test bed. Based on the measurements, a Semi-Markov process (SMP) is introduced to model the behavior of software life-extension and to estimate the availability and performance measures.

A. Test configuration

We use the same test bed as described in Section III-C. A VM is created with 512MB of memory and memcached is launched with the maximum limit set to 900MB. The VM may crash due to out-of-memory with this configuration if the number of inserts exceeds certain limit and no additional memory is allocated to the VM. We implement a script which applies software life-extension automatically when the VM is about to run out of free swap space. The script runs on a host server and monitors the free swap space of the VM every ten seconds using Simple Network Management Protocol (SNMP). When the amount of free swap space becomes smaller than or equal to 1 MB, the script invokes Xen’s command to allocate 64MB of additional memory to the VM. If the command is carried out successfully, the lifetime of the VM is prolonged. Regardless of whether life-extension is applied or not, the VM eventually fails due to out-of-memory. The failed VM is detected manually and a recovery operation is carried out accordingly. The recovery operation includes the destruction of failed VM, creation of a new VM with the same VM image, and start of memcached and some monitoring scripts.

B. Clients

Two client programs for memcached are developed for the experiments. A client program, measurement client, is aimed to measure the availability of memcached, while the other client, load client, is used for accelerating the aging by imposing numerous insert operations. The load client requests a “put” operation which inserts 1MB of data to the memcached by a key that is randomly sampled from the set $S_{key}$, which contains 10000 unique keys. The requests are generated in the way that the memcached sees Poisson arrival with rate $\lambda_{load}=10 [1/sec]$. On the other hand, the measurement client requests a “get” operation which looks up the cached data in the memcached by a specific key that is sampled from the subset $S'_{key} \subset S_{key}$ which contains 1000 unique keys. As $|S'_{key}|$ is one tenth of $|S_{key}|$, we assume here the locality of cache access. With a certain probability, the request encounters a cache miss upon which the client subsequently requests a “put” operation to insert 1MB of data. The time between requests is sampled from exponential distribution with rate $\lambda_{cache}=1 [1/sec]$. The measurement client executes continuously even when the memcached is not available so that it counts the number of request drops during VM failure.
TABLE V. Observed VM lifetimes by ten times of life tests

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Applied</td>
<td>71</td>
<td>1550</td>
<td>348</td>
<td>189</td>
</tr>
<tr>
<td>2</td>
<td>Not applied</td>
<td>70</td>
<td>329</td>
<td></td>
<td>169</td>
</tr>
<tr>
<td>3</td>
<td>Not applied</td>
<td>74</td>
<td>326</td>
<td></td>
<td>230</td>
</tr>
<tr>
<td>4</td>
<td>Not applied</td>
<td>75</td>
<td>327</td>
<td></td>
<td>219</td>
</tr>
<tr>
<td>5</td>
<td>Applied</td>
<td>76</td>
<td>1440</td>
<td>318</td>
<td>201</td>
</tr>
<tr>
<td>6</td>
<td>Applied</td>
<td>72</td>
<td>1518</td>
<td>330</td>
<td>211</td>
</tr>
<tr>
<td>7</td>
<td>Applied</td>
<td>81</td>
<td>1512</td>
<td>347</td>
<td>180</td>
</tr>
<tr>
<td>8</td>
<td>Applied</td>
<td>72</td>
<td>1494</td>
<td>341</td>
<td>177</td>
</tr>
<tr>
<td>9</td>
<td>Applied</td>
<td>79</td>
<td>1568</td>
<td>339</td>
<td>177</td>
</tr>
<tr>
<td>10</td>
<td>Not applied</td>
<td>72</td>
<td>363</td>
<td></td>
<td>217</td>
</tr>
</tbody>
</table>

**C. Measurement results**

We carried out repeated life time tests in each of which a memcached starts on a VM, the VM fails due to out-of-memory, and a new VM is created for the next test. The lifetime events such as memcached startup, start swapping, VM failure, software life-extension are recorded in log files and the time to start swap (TTS), the time to VM failure (TTF), the time to apply software life-extension (TTL), and the time spent for manual VM recovery (TTR) are computed from the log. The observed VM lifetimes by ten times of tests are shown in TABLE V.

Software life-extension is applied successfully six times in the experiments and it extends the lifetime of the VM more than four times when life-extension is applied. In these experiments, the success probability of software life-extension mainly depends on the coverage of detection of critical states. If we decrease the monitoring interval to observe the free swap space more frequently, the coverage might be increased. The detection coverage, in this experimental study, is roughly estimated 60% from the empirical data. Let TTF, be the TTF observed in the life test i and TTL, be the TTR in the life test i, respectively. The empirical availability of the VM is computed as:

\[
A_L = \frac{\sum_{i=1}^{n} TTF_i}{\sum_{i=1}^{n} (TTF_i + TTR_i)} = 0.841091.
\]

The availability without software life-extension can be estimated by counting the test results for which software life-extension is not applied:

\[
A_N = \frac{\sum_{i=2,3,4,10} TTF_i}{\sum_{i=2,3,4,10} (TTF_i + TTR_i)} = 0.616972.
\]

As can be seen, software life-extension greatly improves the availability of the VM because of the difference in TTFs. Since the tests are carried out under the accelerated workload by the load client, the observed TTFs are considerably shorter than the TTFs in real life. The potential availabilities are much higher than the computed \(A_L\) and \(A_N\) according to the TTFs under the real workload.

**D. Availability model**

From the experimental results, we present a SMP model that describes the general behavior of software life-extension and estimates the availability of VM. As studied in Section III-C, out-of-memory failure occurs only after starting memory swapping. The state, when a VM is using swap space, can be considered as a failure probable state in a software aging model. Therefore, we present a four states SMP model which represents the behavior of software aging and life-extension as shown in Figure 6. The model is analogous to the availability model presented in Section II-D. Unlike the simple CTMC model shown in Figure 1(c), the SMP does not limit the distributions of state transition times to be exponential distribution and hence we can model the system behavior more precisely.

The VM starts in an available state which is represented by State 0. When the VM starts using swap space, the state changes to State 1. If software life-extension is performed successfully before the VM failure, the state changes to State 2. Regardless of whether software life-extension is performed or not, the VM eventually fails and enters in State 3. After manual recovery operation, the state is returned to State 0. Although the VM instance after the recovery operation is not identical to the failed VM, we consider them in a single SMP model under the assumption that each VM instance follows the same state transition.

The SMP can be specified by a state transition probability matrix \(P = \{p_{i,j}\}_{0 \leq i, j \leq 3}\) and sojourn time distributions \(H_i(t), 0 \leq j \leq 3\). Let \(c\) be the coverage of software life-extension (i.e., the success probability of life-extension), the transition probability matrix is given by

\[
P = \begin{bmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & c & 1 - c \\
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0
\end{bmatrix}.
\]

Consider the steady-state probability vector \(v = [v_0, v_1, v_2, v_3]\) which satisfies the equations \(v = vP\) and \(\sum_j v_j = 1\). Solving the system of equations, we get

\[
v = \begin{bmatrix}
\frac{1}{3 + c} \\
\frac{c}{3 + c} \\
\frac{1}{3 + c}
\end{bmatrix}.
\]

Using the two-stage method [9], the steady-state probability vector \(\pi = [\pi_0, \pi_1, \pi_2, \pi_3]\) of the SMP is computed by
\[ \pi_j = \frac{v_j h_j}{\sum_{k=0}^{3} v_k h_k}, \quad j \in \{0,1,2,3\}, \]

where \( h_j \) represents the mean sojourn time in State \( j \) defined:

\[ h_j = \int_0^\infty (1 - H_j(t)) \, dt, \quad j \in \{0,1,2,3\}. \]

Now we fit the measurement results shown in TABLE V to the SMP model to estimate the steady-state availability. First, the coverage of software life-extension is determined to be 0.6 which is the detection coverage. Regarding the sojourn time distributions, TTS and TTL correspond to the sojourn times in State 0 and State 3, respectively. We compute the sojourn times for State 1 by

\[
\begin{align*}
(TTF_1 - TTS_i) & \quad i = 2,3,4,10 \\
(TTL_1 - TTS_i) & \quad i = 1,5,6,7,8,9,
\end{align*}
\]

where TTS, and TTL, represent the TTS and TTL observed in the life test \( i \), respectively. The sojourn time in State 2 is computed by \( TTF_i - TTL_i, i = 1,5,6,7,8,9 \). Although we do not know the exact distributions of sojourn times, the mean sojourn time \( h_i \) can be estimated by sample mean. The estimated mean sojourn times are computed

\[ \hat{h} = [\hat{h}_0, \hat{h}_1, \hat{h}_2, \hat{h}_3] = [74.2, 262.6, 1176.5, 197.0]. \]

Since the VM is available in the all states except State 3, the steady-state availability is given by

\[ \hat{A}_L = \sum_{j=0}^{2} \pi_j = \frac{\sum_{j=0}^{2} v_j h_j}{\sum_{k=0}^{3} v_k h_k} = \frac{h_0 + h_1 + c \cdot h_2}{h_0 + h_1 + c \cdot h_2 + h_3}. \]

Substituting the variables with the estimated coverage and the mean sojourn times \( \hat{h} \), we get the estimated steady-state availability \( \hat{A}_L = 0.84064 \). The result fits well with the empirical availability \( A_L \) as seen in the previous section C.

**E. User-perceived metrics**

Next, we analyze the user-perceived availability and performance. TABLE VI shows the statistics obtained from the measurement client during the ten times of life tests on memcached.

The request arrival rate can be computed by the total number of accesses divided by the total test time \( T=12398 \) [sec]. However, the computed request arrival rate 0.9545 disagrees with the expected request rate \( \lambda_{\text{mes}}=1.0 \) generated by the measurement client. This is caused by the implementation of the measurement client which employs *closed system model* [11] for request generation. In a closed system model, new request is generated only after the completion of the precedent request. During the period of the VM failure, our measurement client waits for the request time-out which results in a delay of the subsequent request. To get a better analysis, we need to separate the arrival rate during the VM up states from that in the VM down state.

![Figure 7. Cache hit rate during life test 1](image)

Considering the separation, we can compute the effective request arrival rates as shown in TABLE VII. The request arrival rate in the VM up states \( \lambda_{\text{up}} \) is close to the original request rate \( \lambda_{\text{mes}}=1.0 \).

![Table VII. Number of requests from measurement clients](image)

<table>
<thead>
<tr>
<th>Test No.</th>
<th>Num. of requests</th>
<th>Num. of request processed</th>
<th>Num. of request drops</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1729</td>
<td>1554</td>
<td>175</td>
</tr>
<tr>
<td>2</td>
<td>467</td>
<td>333</td>
<td>134</td>
</tr>
<tr>
<td>3</td>
<td>515</td>
<td>330</td>
<td>185</td>
</tr>
<tr>
<td>4</td>
<td>477</td>
<td>320</td>
<td>157</td>
</tr>
<tr>
<td>5</td>
<td>1599</td>
<td>1429</td>
<td>170</td>
</tr>
<tr>
<td>6</td>
<td>1675</td>
<td>1507</td>
<td>168</td>
</tr>
<tr>
<td>7</td>
<td>1599</td>
<td>1449</td>
<td>150</td>
</tr>
<tr>
<td>8</td>
<td>1549</td>
<td>1420</td>
<td>129</td>
</tr>
<tr>
<td>9</td>
<td>1698</td>
<td>1566</td>
<td>132</td>
</tr>
<tr>
<td>10</td>
<td>526</td>
<td>373</td>
<td>153</td>
</tr>
</tbody>
</table>

Total: 11834 10281 1553

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda_{\text{up}} )</td>
<td>0.992579</td>
<td>Request arrival rate in VM up states</td>
</tr>
<tr>
<td>( \lambda_{\text{down}} )</td>
<td>0.789503</td>
<td>Request arrival rate in VM down state</td>
</tr>
</tbody>
</table>

![Table VIII. Effective request arrival rates in VM up states and VM down state](image)

**F. User-perceived measures**

Regarding the number of access, we can compute the effective number of request drops during the life tests on memcached.

![Table IX. Observed cache hit rates](image)

<table>
<thead>
<tr>
<th>Test No.</th>
<th>Life extension</th>
<th>Num. of cache hits</th>
<th>Cache hit rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Applied</td>
<td>1469</td>
<td>0.945302</td>
</tr>
<tr>
<td>2</td>
<td>Not applied</td>
<td>264</td>
<td>0.792793</td>
</tr>
<tr>
<td>3</td>
<td>Not applied</td>
<td>266</td>
<td>0.806061</td>
</tr>
<tr>
<td>4</td>
<td>Not applied</td>
<td>260</td>
<td>0.8125</td>
</tr>
<tr>
<td>5</td>
<td>Applied</td>
<td>1319</td>
<td>0.923023</td>
</tr>
<tr>
<td>6</td>
<td>Applied</td>
<td>1393</td>
<td>0.924353</td>
</tr>
<tr>
<td>7</td>
<td>Applied</td>
<td>1336</td>
<td>0.922015</td>
</tr>
<tr>
<td>8</td>
<td>Applied</td>
<td>1304</td>
<td>0.91831</td>
</tr>
<tr>
<td>9</td>
<td>Applied</td>
<td>1448</td>
<td>0.924649</td>
</tr>
<tr>
<td>10</td>
<td>Not applied</td>
<td>311</td>
<td>0.83378</td>
</tr>
</tbody>
</table>

![Figure 8. Observed cache hit rates](image)
Using the result of estimated steady-state availability with $T=12398$, we obtain $N_{\text{drop}} = 1555.32$, which gives a good estimate of the total number of request drops in TABLE VI.

The user-perceived performance can be characterized by cache hit rate of memcached. The observed number of cache hits and cache hit rates are shown in TABLE VIII. As shown in the difference of cache hit rates, the performance is improved by software life-extension (i.e., higher hit rates in the tests 1 and 5-9). In contrast, if we apply software rejuvenation to memcached, all of memory content is cleared at the rejuvenation and it causes considerable cache misses after restart. Software rejuvenation is not preferable in terms of user-perceived performance in this scenario. Figure 7 presents the changes of cache hit rate in the life test 1. The cache hit rate increases over 90% when the lifetime is longer than 650 seconds. If software life-extension is not applied, the lifetime ends after approximately 330 seconds (as observed in TTFs in Table V) and the cache hit rate does not reach 90%.

V. DISCUSSION

While the experiments show the feasibility and effectiveness of software life-extension by dynamic memory allocation to a VM, the approach of software life-extension is not limited to this experimental scenario. As discussed in Section II-C, workload control is an alternative approach for extending the lifetime of software suffering from aging. As we examine the relationship between the amount of memory allocation and the lifetime in Section III, for workload control approach, the relationship between workload and lifetime needs to be analyzed. Characterization of workload-aging relationship [6], accelerated degradation tests [12] or accelerated life tests [13] can be used for this purpose.

The VM-based software life-extension appears to be generally applicable to any kind of applications running on VM. However, it limits the applications that can recognize dynamically added memory and make use of them. For example, applications running on Java Virtual Machine (JVM) cannot take the advantage of dynamically added memory because JVM max heap size is set during initialization and it cannot be modified during the execution.

The problem of configuration error on memcached pointed out in our experiments may be removed by allocating sufficiently large amount of memory to VM or by reconsidering the configuration of memcached. However, the trouble shooting for such configuration errors is not simple as with debugging in software development. We need to perform experiments to reproduce the problem and figure out how large amount of memory is necessary for robust operation. In addition, unlike memcached, applications may not provide configuration parameters for limiting the memory usage. Although software life-extension does not remove the source of a problem, it is useful for mitigating the software aging temporarily in system operation.

VI. RELATED WORK

Aging phenomena have been observed in many software products such as Apache web server [12], Linux [15], JVM [16] and telecommunication systems [17]. In previous literature, software aging is considered to be rooted software fault so called aging-related bug. However, as experienced in this paper, software aging can be observed without aging-related bug by another factor in configuration mismatch between application and execution environment. Such misconfiguration is more likely to appear in virtualized environment because user can easily modify resource allocation even during the runtime.

Many studies mention that aging phenomenon is closely related to the workload on the software [5][6][18]. Bovenzi et al. presented a general procedure to characterize the workload-aging relationship by set of experiments [6]. In order to reduce the time to obtain the lifetime of software, Matias Jr. et al. applied accelerated degradation tests [12]. The method is useful to obtain lifetime distribution in IPL-Weibull or IPL-lognormal model in a short time, and it has been applied to characterize software aging in an on-line application system [19]. The workload-aging relationship is relatively simple in our experimental study, but real workload characteristic of memcached highly depends on the application as studied in [20]. In practical system, workload-aging characterization needs to be performed first for planning software life-extension.

The studies of software aging and rejuvenation have been broadly classified into two categories; model-based and measurement-based. The recent survey pointed out that many works are devoted to analytic modeling for rejuvenation scheduling and advocated the necessity of the hybrid (i.e., model and measurement based) approaches [21]. The argument should be true for software life-extension as well. Our experimental study is based on measurements and the observed results are analyzed with analytical model that is an extension of outcomes from model-based study [2][8][22]. While we do not provide the analysis of an optimal trigger for software life-extension in this paper, our preliminary study might open further research direction of the optimal scheduling analysis (e.g., combination of software rejuvenation and life-extension).

Since software life-extension allows software to continue its execution even under software aging states, the concept has a similarity to failure-oblivious computing [23] which continues the software execution even after system error by neglecting the error. Google admits the effectiveness of failure-oblivious computing in their parallel data analysis with Sawzal [24]. Similarly, Rx is presented as a safe survival technique for software failures using checkpoint and re-execution in a modified environment [25]. In contrast to these failure survival techniques, in this paper we focus on software life-extension as a preventive maintenance operation to postpone the time to failure occurrence.

Clustering is the most commonly applied solution of tolerating failures of cache server. Web cache servers are often configured as clusters for high-performance, scalability and availability [26]. Dynamo [27] and PNUTS [28] keep some essential data in memory and create replicas to be distributed to multiple locations. Recently RAMCloud has presented a quick recovery technique for DRAM-based storage system by scattering backup data across several disks.
and harnessing hundreds of servers to reconstruct the lost data [29]. While clustering or replication requires a careful design and deployment of system with network configuration, our solution provides a simple countermeasure to a failure of cache server that can be performed manually in the operation.

VII. CONCLUSION

We have presented a new countermeasure to software aging which extends the lifetime of software execution before encountering a failure caused by resource exhaustion. The feasibility of software life-extension is studied in the scenario of dynamic resource allocation to a virtual machine hosting a memcached which suffers from memory aging due to wrong configuration of maximum memory consumption. We have observed that the lifetime of memcached was greatly improved even by a small amount of additional memory allocation. Compared with software rejuvenation, software life-extension is especially preferable for the applications storing essential data in memory storage because it keeps memory content as long as possible during the lifetime. Through the experiments on memcached, we studied the effectiveness of software life-extension in terms of system availability, the number of request drops and cache hit rate.

ACKNOWLEDGMENT

The authors would like to thank Kishor Trivedi for his technical advice on modeling, and our shepherd, Aleksandar Shurbevski and Lada Ducheckova, for their valuable feedback on this paper.

REFERENCES


