A Feedback Control Approach for Preventing System Resource Exhaustion Caused by Software Aging

Yun-Fei Jia[1], Zhi Quan Zhou[2], Renbiao Wu[1]

[1] Tianjin Key Laboratory for Advanced Signal Processing, Civil Aviation University of China, China
[2] Institute of Cybersecurity and Cryptology, School of Computing and Information Technology,
University of Wollongong, Australia
yfjia@cauc.edu.cn, zhiquan@uow.edu.au, rbwu@cauc.edu.cn

Abstract

The dynamic behavior of software systems attracts widened attention through the phenomenon of software aging. Software aging is caused by runtime environment deterioration, such as the gradual loss of memory or CPU cycles. The dynamic behavior of aged software systems can be described by a set of evolving resource variables, including CPU usage, I/O bandwidth, available memory and the like. From this point of view, an aging software system can be analogous to a dynamic system. Control theory provides sound and rigorous mathematical principles to analyze dynamic systems and build controllers for them. This paper introduces control theory to analyze and build a control model and apply control techniques to an aged web server. First, we treated the software system as a black box, and conducted controlled experiments to build the relationship between input and output. Then, these input-output couples are used to build a control model via a system identification method. Finally, a PI (proportional-integral) controller is designed to adjust the aged state of the software system, and software rejuvenation techniques are customized to target the web server. Performance testing shows that our approach can accurately track the reference value set by the website administrator.

Keywords: Software aging, MIMO control, System identification, Software rejuvenation, Feedback control

1 Introduction

The wide-spread service-based software calls for high availability. Nevertheless, when an application server executes continuously for long periods of time, it may accumulate many error conditions or uncollected garbage in its process space. Consequently, the process will have insufficient computing resources to respond to the client's requests. From this perspective, software aging can be attributed to gradual runtime environment degradation, caused by aging-related bugs [1-2], where a process cannot get sufficient computing resources. Eventually, the computer system will crash or hang when all computing resources are exhausted.

Software rejuvenation can clean runtime environment before severe aging occurs. Commonly used techniques are to reset the whole computer system or to restart the software system when aging signs are detected. Most literature focuses on the optimal timing of rejuvenation, and the proposed rejuvenation techniques are usually "heavyweight", i.e., they usually involve termination of service [2-5]. However, under some conditions, service downtime will incur tremendous economic loss, and some safety-critical services cannot be reset.

To avoid crashes, the available computing resources must be retained to some extent. Blindly limiting the resource usage will also incur too much cost, such as longer response time or lower throughput. Lightweight control policy is expected to retain enough available computing resources at the expense of lowering the capacity of the computer system. In fact, lightweight rejuvenation techniques have attracted more attention in recent years, Machida et al. [6] proposes a life-extension technique to maintain availability of an aged virtualized system. A two-level rejuvenation policy is proposed to reduce the downtime of rejuvenation [7], which is improved by Ning et al. [4]. Kourai and Ooba [8] proposes a type of lightweight rejuvenation technique called "zero-copy migration", which can relocate aged VMs to a clean virtualized system without any copying. Yan et al. [9] quantitatively studies the relationship between resource consumption and changing workload. The obtained results can help an administrator tune the system properly to reduce business losses incurred by software aging problems.

Although these lightweight rejuvenation techniques incur much less downtime, they still have several limitations. For example, Ning et al. [4] and Xie et al. [7] require some assumptions on the arrival rate of memory leaks, which hinders the application of their

*Corresponding Author: Yun-Fei Jia; E-mail: yfjia@cauc.edu.cn
DOI: 10.3966/160792642019092005017

approaches. Furthermore, only one aging indicator (memory leak) is considered in some studies [6, 8-9], although more resources usually interact with each other during the aging process.

There has been increasing research efforts in applying control theory to behavior management for web servers, databases and storage systems [10-11]. [12] provides an additional component to know the nature of application, so that the performance degradation, as a results of sudden rise of workload, will be mitigated. [13] proposes a new algorithm for multi objective task scheduling in cloud. [14] proposes a GA-CAS algorithm to address multiple objective optimization problem in cloud system. Inspired by those studies, this paper intends to implement a lightweight no-downtime rejuvenation technique covering more aging indicators. In this paper, a controlled experiment is conducted to study the relationship between these parameters and resource usage of a web server, Apache httpd (referred to as Apache hereafter). Based on the experimental data, we propose a linear model that approximates the relationship between software aging and several important parameters of Apache. Based on the MIMO (multiple-input and multiple-output) model, we have designed a PI (proportional–integral) controller to online-control the resource variables of concern below a specified threshold. Doing so will work around the consequence of software aging with the target of not stopping the server.

Since rejuvenation techniques usually have close relationships with subject software, this paper investigates the operating mechanism of a typical web server–Apache. Most web servers provide certain parameters for users to customize the system at different runtime environments. Some of these parameters may significantly affect the performance of the software system and the usage of system resources [15]. Adjusting these parameters can proactively avoid resource exhaustion resulted from software aging. More specifically, limiting the number of concurrent threads or processes of a web server can mitigate the effect of memory leakage. The consequential cost is capacity shrinkage, i.e., only partial requests can be handled and responded to. In addition, when the workload of a web server is heavy, i.e., many requests are incoming, the response time of all requests will increase to some extent, and some of the requests will be rejected by the web server. This is because most web server applications work based on a best-effort service model. Web servers have to dispatch and reject those redundant requests. This process will also come at the cost of some CPU cycles and memory. We have to control the number of requests inputted into the listening queue of the web server. This paper intends to study the relationship between resource usage and parameter setting of target software systems.

Compared to existing lightweight rejuvenation techniques, our approach has several advantages:

(1) Several aging indicators can be incorporated in our approach, such as free physical memory and average load;

(2) No assumption on aging indicators (such as arrival rate and memory leak speed) is required, which enables our approach to be applicable to more target software systems;

(3) Our approach can be used as a framework to easily incorporate more rejuvenation techniques such as memory add [6] and VM migration [8].

The rest of the paper is organized as follows. Related studies are presented in Section 2. Section 3 reports the experiments that study the relationship between software aging and related parameter settings. A system identification method is employed to quantitatively build a MIMO model in Section 4. This model can quantitatively describe the relationship between parameter settings and resource usage. A PI controller is designed by a pole assignment method in Section 5. In Section 6, our proposed controller is evaluated in terms of fastness and accuracy, and the cost of rejuvenation is also evaluated. Section 7 concludes this paper.

2 Related Work

Aging modelling and control are key questions throughout two decades of software aging study [16]. Aging control study intends to seek cost-effective techniques to clean the aged software system. Because rejuvenation will usually incur larger cost, some newer rejuvenation techniques are customized for the subject software of concern.

[17] studied the abnormal behavior propagation mechanism in networked software. Xie et al. proposed a two-level rejuvenation policy based on the degree of aging, i.e., when the subject software is slightly aged, only some modules are restarted, and when it is severely aged, the whole software should be restarted [7]. Zhao et al. [18] constructed a model to ensure the performance of the Apache HTTP server. Their conclusions are based on an in-depth analysis of the working mechanism of the target software system. With the wide-spread acceptance of Cloud Computing, the aging phenomenon in virtual machine is well reported, and specific rejuvenation techniques are implemented. The rejuvenation techniques for virtual machines can allow for little downtime, which is implemented by suspending and resuming the aged instances [3]. Rejuvenation of aged instances can also be implemented by mitigating it [19]. This technique can incur little throughput loss and no downtime.

There is an increasing trend for using control theory in performance guarantee. It is usually implemented by monitoring the performance or predicated workload of a web server, which are fed back to a “controller” to compute whether it has crossed a specified threshold. If it has, some control techniques will be invoked, such as rejecting some requests, reducing the video frame rate (for a video website), and so on. In fact, this method forms a closed-loop to control the performance of a website. Key et al. [20] reported the method and the implementation of service differentiation. Service differentiation provides some ideas about controlling the performance of a web server. When the web server is overloaded, only important parts of requests will be handled, and others are discarded with the purpose of ensuring the quality of service (QoS) of some VIP clients. A differentiated caching mechanism is proposed to multiple levels of service in proxy caches [21]. They proposed a control-theoretical approach to enforce performance differentiation on information access. The feedback control principle is widely used in QoS guarantee, although some literature does not explicitly highlight this.

However, control theory application in software aging has received much less attention. This paper intends to lay well-understood theoretical foundations for software aging control and seek applicable techniques to prevent resource exhaustion.

3 Experimental Set-up

3.1 Software Experiment Methodology

It should be noted that there is a difference between an empirical study and a controlled experiment. According to Vaidyanathan and Trivedi [1], an empirical study usually makes an observation of, and collection of data from, the running real world computer system. In an empirical study, the experimental conditions are not controlled by the researcher. The advantage of an empirical study lies in that it can reflect the activity of a computer system in a real working scenario. However, the disadvantage lies in that we can hardly find the causes of the observations, because the working condition of the concerned software is not known. On the contrary, a controlled experiment can adjust the working condition of the subject software, whilst we must try to make the working conditions or runtime profile adhere to its real-world working conditions. In this paper, controlled experiments are conducted in order to probe the mechanism of software aging.

The subject software should be adopted carefully. In a software aging experiment, the subject software should be mature and widely applied in the real-world. The subject software must be selected in such a way that the resulting observations can make sense. In our experiment, Apache is employed as our subject software. Apache is the most popular web server used on the Internet. The aging phenomenon of Apache 1.3 had been reported by Grottke et al. [22]. The current stable version of Apache is 2.0, which is in use by many websites. Although the newest version of Apache httpd is 2.4, the 2.0 version is still widely used in the world. In addition, the Apache 2.0 has mature and similar features as the 2.4 version. Hence, we select Apache 2.0 as our subject software. In comparison to Apache 1.3, Apache 2.0 is enhanced greatly in many respects, including Unix threading support, multiprotocol support, updated regular expression library, etc [15]. Thus, software aging in Apache 2.0 should be revisited.

3.2 Experimental Design

The experimental set-up is the same as that in our previous work [23], briefly described as follows: In our experiments, Apache is deployed on a workstation. This computer is used as a web server in our experiments. Three other workstations with the same hardware configuration are used as clients to generate artificial concurrent requests to access the web pages on the Apache server. All four workstations are connected via a switch. A web server test tool, httperf [24], is deployed on the clients to generate artificial connection requests for static html pages with exponential time intervals to the web server. System resource variables are collected by DataCol, a shell script written by us, including free physical memory (memFree), average load (loadAvg), etc. The resources variables can reflect the memory consumption and CPU usage. In Linux OS, loadAvg refers to the number of processes waiting for CPU or disk I/O [25]. The vector (memFree, loadAvg) can be used as the output of the control model.

Apache has many parameters influencing software aging in terms of resource usage. The parameter MaxClients has a direct relationship with the accumulated effects of the residual defects in the system [15]. Hence, we can implement a new rejuvenation technique, i.e., limit the resource consumption of Apache by adjusting the value of MaxClients. Nevertheless, when too many requests are incoming, limited child processes cannot handle and respond to the requests in time. Thus, we must reject some incoming requests according to a prespecified policy. In this paper, both connection rate (conRate) and MaxClients will be included in the model as input.

In the next step, we should build the relationship between input and output, since our target is to limit the value of output by adjusting the input. Because the dynamic behavior and working mechanism of Apache is much too complex, we cannot build a mathematical control model based on its physical principle; instead, such complex systems can be treated as a black box, and control models can be built through a system identification method. This method demands massive input/output couple data to estimate their relationship. Hence, we need to adjust the input and record the corresponding output repeatedly to get sufficient input/output observations.

3.3 Online Parameters Control

Apache is a multiple parallel processing web server. It will spawn a number of child processes to handle the accepted requests in parallel. The parent process of Apache will not handle requests but rather monitor the status of the child processes. More specifically, when the workload is lower, it will kill some excessive idle child processes to release memory. When workload surges, it will spawn more child processes to increase its processing capacity. This feature can be used to control the resource consumption thus retaining some resources for other co-located applications. When Apache starts, the default number of spawned child processes is specified in the configuration file and read into scoreboard. Scoreboard is a piece of shared memory which can be read/written by the parent process. In this paper, we modify the source code of Apache, and specify the value of MaxClients by an external module called Apache controller, a tool implemented by us. More specifically, Apache controller can write the value of MaxClients into scoreboard. The parent process will repeatedly read the scoreboard and then set the number of child processes, thus resource usage of Apache will be adjusted appropriately.

When a request is incoming, Apache will first buffer it in its listening queue, then the parent process will dispatch this request to a child process. In our experiment, the arriving requests sent by httperf are intercepted by the Apache controller. In this way, the number of requests to arrive and to be handled by Apache can be controlled. The working mechanism of the controlled Apache is illustrated in Figure 1.

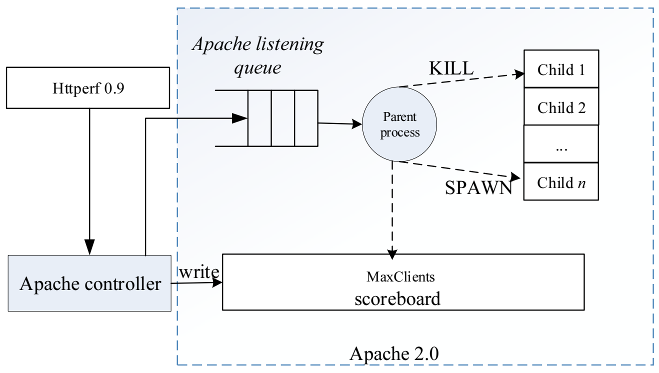

Figure 1. Online parameters control

In Figure 1, mass artificial requests generated by httperf are sent to the Apache controller. Some of those will be discarded so that the number of remaining requests arriving in the listening queue can be controlled as we specified. The parent process will dispatch each request to an idle child process. In addition, the parent process will poll the scoreboard for the status of child processes and some parameters including MaxClients. When MaxClients changes, the parent process will kill or spawn some child processes accordingly.

Our approach can be implemented by a typical feedback control system, as described in Figure 2.

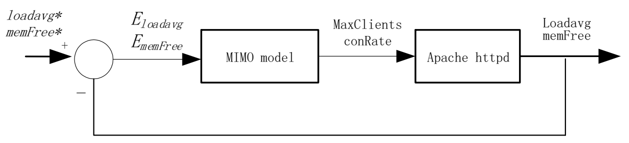

Figure 2. Feedback control of apache

In Figure 2, the expected values of available resources (loadavg, memFree) are set by the administrator, and the collected resources are compared against the expected values, with the error as input into the MIMO model. Accordingly, the parameters of Apache (Maclients, conRate) will be adjusted by the output of the MIMO model.

To summarize, our approach can be implemented as follows:

(1) To identify the relationship between available resources and key parameters of Apache: This can be implemented by a System Identification method.

(2) To monitor the available resources of an operating system: Taking the Linux OS as an example, reading the */proc* can get the free memory and load average values.

(3) When performance degradation occurs, available resources degrade beyond a threshold (which can be set by the administrator), a light rejuvenation will be triggered.

(4) The light rejuvenation proposed in this paper can be implemented by adjusting the parameters of Apache, i.e., MaxClients and conRate. The adjusted quantity can be calculated based on the relationship built in Step 1).

4 System Identification

Control theory and control engineering focuses on model building and controller design. All techniques for analysis and design of a control system are based on an appropriate control model. Those mathematical models can describe the dynamics of a real or virtual system. Some models can be built based on the working mechanism of a physical system. These systems are usually small and simple. If the working mechanism is not clear or not well understood, a system identification technology can be employed to build a control model.

In this section, following the method of system identification [26], we constructed a linear MIMO model of the relationship between resource consumption and parameters setting of Apache. The system identification technology treats the system under consideration as a black box, and establishes a mathematical representation of the physical system from experimental data. Such a model is some form of pattern that explains the observed experimental results and allows for predictions of future system responses to be made.

The quality of system identification partially depends on the quality of the inputs, which are under the control of the systems engineer. In order to trigger the dynamics of Apache, the input design should maximize the coverage of the entire input space. Using a testing method reported in our previous study [23], we conducted capacity testing and found that, when MaxClients was set to 250 or larger, almost all physical memory would be in use; if it was set larger than that, the swap space would be used, and would deteriorate the system's capacity. We further found that Apache had a capacity of 320 requests per second. Figure 3 shows the input values of MaxClients and conRate.

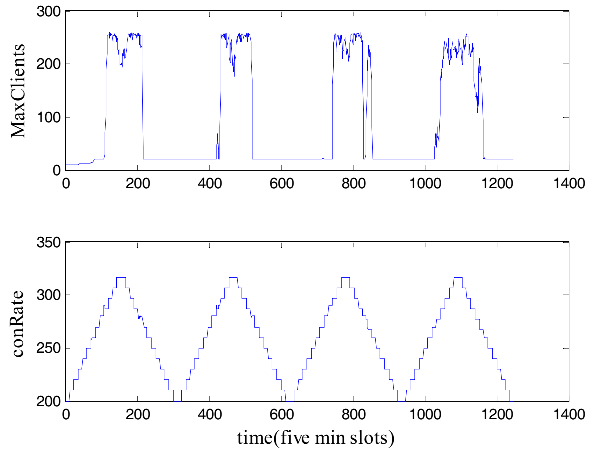

Figure 3. Input in experiment

A typical discrete form of a MIMO control model can be expressed by the following equations:

$$\begin{aligned} x(n+1) &= Ax(n) + Bu(n) \\ y(n) &= Cx(n) \end{aligned} \tag{1}$$

Where n indexes time; in our case, y is a 2×1 vector which indexes output; x is an $m\times m$ matrix which indexes the state of the target system, and u is a 2×1 vector which indexes the input. More specifically, the input of the model is $u(n)$ = (MaxClients, conRate)^T, and the output is $y(n)$ = (loadAvg, memFree)^T. $x(n)$ is the intermediate variables of the server and $\boldsymbol{x}(0)$ can be set to vector 0. $x(n)$ represents a state-space model of m dimensions (where m can be an arbitrary integer). Usually, the higher m is, the more accurate the model is. However, when m is too large, it will bring an over-fitting problem that prejudices the generalization of the model. Using a trial and error method, we got the best trade-off when m is set to 2. The dimension m of $x(n)$ in equations (1) represents the complexity of our model. This tells us that the relationship between the input (MaxClients, conRate) and the output (loadAvg, memFree) can be described by a simple model. Nevertheless, our approach can be generalized to a more complex model when the dimension m is large.

Equations (1) cannot be directly estimated by experimental data. We must eliminate the intermediate variable x by Z-transform. Z-transform is one of the mathematical tools used for the analysis and design of discrete-time control systems. The role of Z-transform in digital control systems is analogous to that of the Laplace transform in the continuous-time control systems. Applying Z-transform to the upper equation in Equations (1), we get:

$$x(n+1) = zx(z) - zx(0). \tag{2}$$

Then $x(z)$ can be solved as follows:

$$(zI - A)x(z) = zx(0) + BU(z) \tag{3}$$

Substituting the above equation into the lower equation in Equations (1), we get:

$$y(z) = zC(zI - A)x(0) + C(zI - A)^{-1}BU(z) \tag{4}$$

Applying the inverse Z-transform to the above equation, we can identify the relationship between $y(n)$ and $u(n)$, which is shown in Equation (5).

$$y(n) = \begin{bmatrix} \dfrac{b_1^{11}z^{-1} + b_2^{11}z^{-2} + \cdots + b_n^{11}z^{-n}}{1 + a_1^1 z^{-1} + \cdots + a_n^1 z^{-n}} & \dfrac{b_1^{12}z^{-1} + b_2^{12}z^{-2} + \cdots + b_n^{12}z^{-n}}{1 + a_1^1 z^{-1} + \cdots + a_n^1 z^{-m}} \\ \dfrac{b_0 + b_1^{21}z^{-1} + b_2^{21}z^{-2} + \cdots + b_n^{21}z^{-n}}{1 + a_1^2 z^{-1} + \cdots + a_n^2 z^{-n}} & \dfrac{b_0 + b_1^{22}z^{-1} + b_2^{22}z^{-2} + \cdots + b_n^{22}z^{-n}}{1 + a_1^2 z^{-1} + \cdots + a_n^2 z^{-n}} \end{bmatrix} \begin{bmatrix} u_1(n) \\ u_2(n) \end{bmatrix} + \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} \tag{5}$$

In equation (5), the parameters a, b, and h need to be estimated from the collected experimental data. In this paper, a least-squares method is used to estimate those parameters. In our case, the first half of the observations are used to estimate the parameters via the least-squares method. Then we use the last half of our observations to validate our model. More specifically, we use the first 600 observations to estimate the above parameters, and use the resulting model to forecast the (*loadAvg*, memFree) values of the remaining 600 observations with the observations as input. Figure 4 shows the last 600 observations (corresponding to the data collected from the 50th hour to 100th hour) and the corresponding estimated values. From Figure 4, it can be observed that the curves of loadAvg and memFree outputted by our model are similar to those of the real data. The error shown in Figure 4 can be attributed to the fact that, even if the number of MaxClients can be accurately set, the size (memory consumption) of each child process may be different. For example, a child process which has processed many more requests will have more memory consumption, because it will buffer into itsprocess space the resources that are recently accessed.

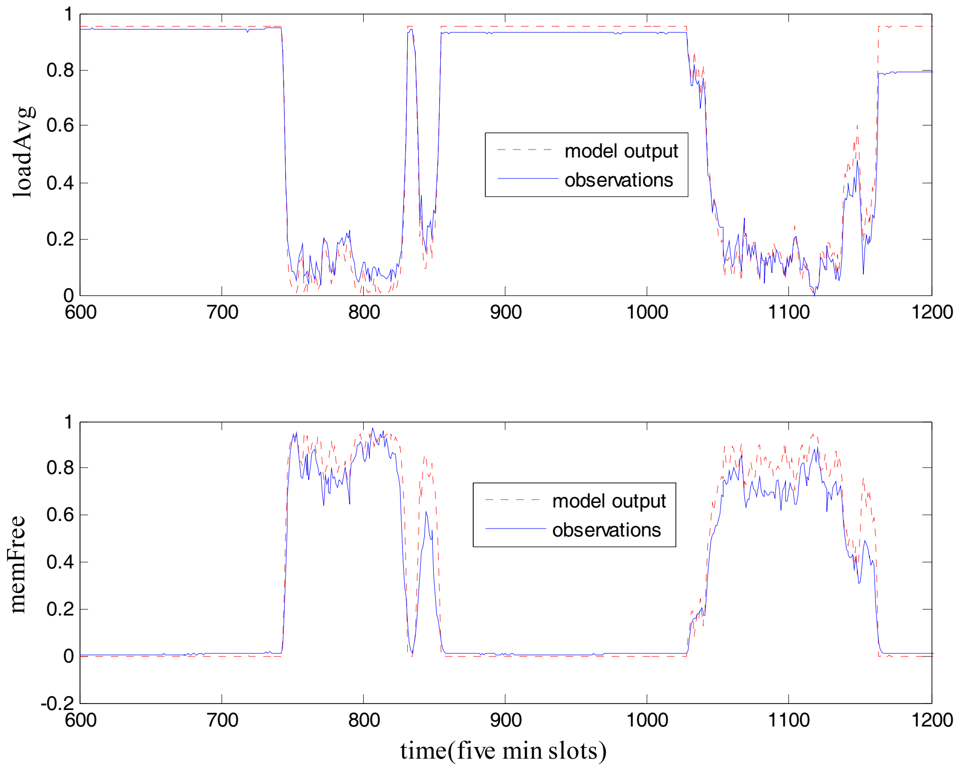

Figure 4. Comparison of experimental observations and forecasted values

5 Design of Discrete Control System

With the control model built in the previous section, we can design a controller to implement our control policy following rigorous control theory. There are several controllers that can be used in the literature [26], such as proportional-derivative (PD) controller, proportional-integral (PI) controller and proportional-integral-derivative (PID) controller. PD controllers can track the expected value quickly, and its weakness lies in larger stable errors. PI controllers can eliminate stable errors and have good robustness, however track the expected value slowly. PID controllers can track expected values quickly and can eliminate stable errors, but is complex and time-costly for the engineer.

When applying control theory to a target system, the adoption of the controller depends heavily on the properties of the target system. In our case, the software system can adjust it parameters quickly, because the inertia of the software system is much less than a real mechanical system. For that reason, we adopt a PI controller to control the Apache web server, and evaluate the fastness and settling time. A typical PI controller operates according to the following law:

$$u_n = K_P e_n + K_I \sum_{j=1}^{n-1} e_j \qquad \textbf{(6)}$$

Where u_n refers to the controllable parameters of *Apache*, and e_n refers to the error between the expected and real values of used system resources. In our case, both K_P and K_I in Equations (3) are 2×2 matrices, and will be designed by a pole assignment method. Usually, K_P can be used to increase the fastness of the control system, and K_P is used to eliminate the steady-state error [21]. The PI controller can be designed as follows:

To track the reference value set by the administrator, we should transform the system model into an error form. Equation (5) can be transformed into a discrete state equation by an inverse Z-transform.

$$Y(n) = AY(n-1) + B_{n-1}(n-1) + B_n u(n) + E_n \qquad \textbf{(7)}$$

Where A, B_{n-1}, B_n and E_n can be calculated from the estimated results of Equation (2).

Let e_1=y_1-y_{1r}, e_2=y_2-y_{2r}, $g_1(n) = \sum_{j=1}^{n-1} e_1(j)$, $g_2(n) = \sum_{j=1}^{n-1} e_2(j)$, $g_3(n)$=$y_1(n)$, and $g_4(n)$=$y_2(n)$, then Equation

(4) can be written as:

$$x(n+1)=\overline{A}x(n)+B_0u(n)+B_1u(n+1)+E$$

Where $\overline{A}$, B_0 and B_1 can be directly calculated from Equation (4), and $E=\begin{bmatrix}-y_1r\\-y_2r\\E_{n1}\\E_{n2}\end{bmatrix}$.

Let $u(n)$=$KG(n)$, we can get:

$$\begin{aligned}G(n+1)&=(1-B_1K)^{-1}(\overline{A}+B_0K)G(n)\\&+(I-B_1K)^{-1}E\end{aligned} \quad \textbf{(8)}$$

where K can be determined by a pole assignment method. The name, 'pole assignment', refers to the fact that the controller is determined in terms of obtaining a closed-loop system with specified poles. The target poles of a PI control system will involve compromises between fastness and steady-state errors. Due to the fact that the software system has little inertia, we pay more attention to eliminating steady-state errors when assigning the poles of Equation (8).

The designed closed-loop control system must be stable. A linear time-invariant system is defined to be bounded-input-bounded-output (BIBO) stable if a bounded input gives a bounded output for every initial value [26]. This definition provides us with a straightforward way to test the stability of our designed system. We set the input to the upper bound and then the lower bound, and observe the outputs. Simulation results validate the designed system as stable.

6 Experimental Results Analysis

6.1 Settling Time

In control theory, the designed controller can be evaluated mainly in two aspects: steady-state error and settling time. Steady-state error accounts for how accurately the control system can track the reference value and settling time measures how fast the control system can track the reference value. Our design goals are twofold: 1) zero steady-state errors; and 2) a small settling time. In the obtained control model, we set two different target values to test the above goals. Step signals are employed in our tests, and the results are illustrated in Figure 5.

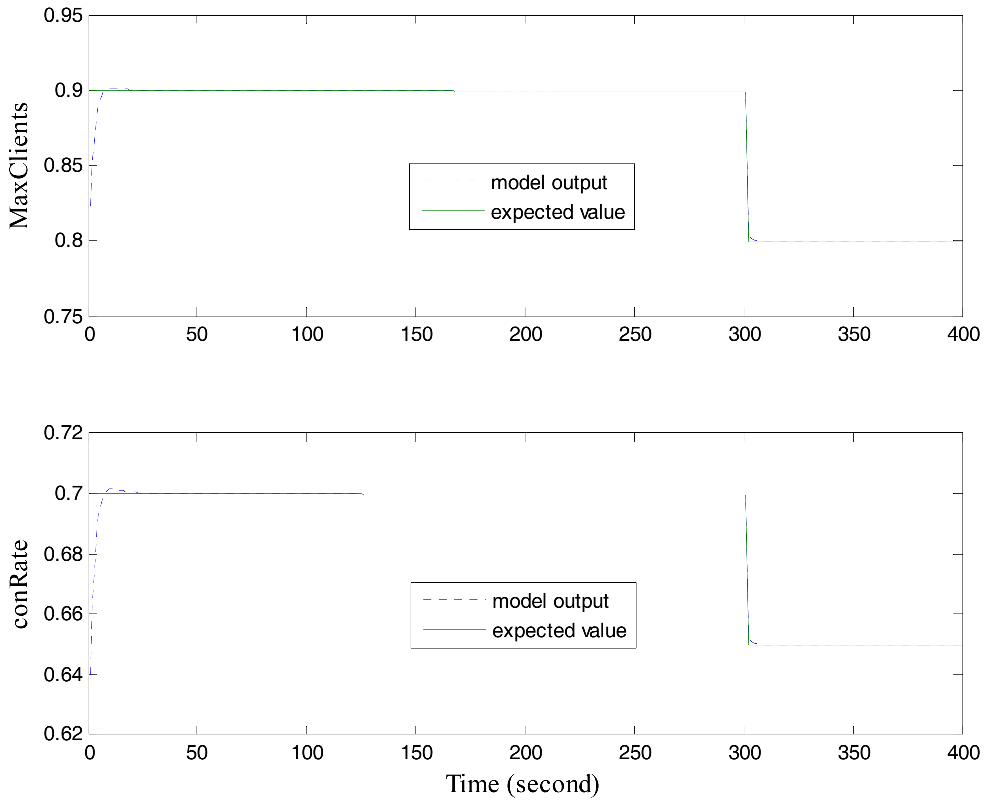

Figure 5. Settling time of PI controller

From the simulation results in Figure 5, we can see that the target values and the output values can hardly be distinguished. This validates that our design can eliminate the steady-state errors accurately. In other words, we can control the resource usage accurately. In addition, from Figure 5 we can see that the settling time is roughly less than 10 seconds, which is fast enough for aging control purposes. This result can also validate the assertion that a software system has little inertia. In Figure 5, there is an obvious difference in settling time when the signal steps up and steps down. When the reference value steps up, there is a larger settling time. This can be explained by the working mechanism of a computer system. Take the curve *MaxClients* in Figure 5 as an example, when it rises, more memory will be allocated, and when it descends, some memory will be reallocated. Because memory allocation will cost more time than cleaning it, there is more settling time when tracking the stepping up signal than the stepping down signal.

6.2 Cost Analysis

Since rejuvenation techniques are closely related with subject software, we compare the performance of our approach against identical software.

Using stochastic model to analyze the reliability of large-scale system is reported in [27]. The aging phenomenon in Apache has been reported by Zhao et al. [18] and Grottke et al. [22]. As aforementioned, Apache employs several child processes to handle the incoming requests. Zhao et al. [18] employs the M/Er/1/K model to simulate its operating mechanism. We try to repeat their simulation with a slight revision, i.e., we simulate aging caused by memory leak as reported by Grottke et al. [22], while Zhao et al. [18] simulate aging signs by service rate degradation. The parameters in our simulation are λ=300reqs/s, μ=320reqs/s, r=2 and K=250. It should be pointed out that K refers to the MaxClients of Apache. The memory leak rate is reported by Abdelmalek et al. [21], i.e., 8.377kB/hr. The steps for our simulation are described as follows:

(1) When a memory leakage is detected, wait until the amount of leakage rises to the amount occupied by one child process of Apache (2MB in our case).

(2) Decrement MaxClients by 1 to mitigate the memory leakage.

(3) Calculate the degraded capacity of Apache, and compare it against the workload (300 reqs/s).

(4) When the degraded capacity is lower than 300 reqs/s, the queue of Apache will rise continuously.

(5) Calculate the rejected requests if the number of requests in the queue is larger than 250.

Following the above steps, we found that rejuvenation is triggered for the first time 238.8 hours from the beginning of the simulation. Then we continue the simulation with a new model, M/Er/1/249, since MaxClients is decremented by 1. This process is continuous until there is a rejected request. The results are listed in Table 1.

From Table 1 we can see that the rejection rate for the first to fifth rejuvenation is 0. This is due to the fact that the capacity of Apache is always higher than the workload. This shows that our approach incurs a lower rejection rate than that in [18]. In addition, the first time request rejection occurs is 1552 hours from the begging of the simulation, which is enough time to back up important data by the administrator.

Table 1. Parameters setting in simulation

Time	Model	Rejection rate
Simulation start	M/Er/1/250 λ=300reqs/s,μ=320reqs/s	0
First rejuvenation 238.8 hours	M/Er/1/249 λ=300reqs/s,μ=320reqs/s	0
	......	
Sixth rejuvation 1552 hours since simulation	M/Er/1/244 λ=300reqs/s,μ=320reqs/s	0.004

Moreover, note that the workload of 300reqs/s employed in our simulation is very heavy, since the capacity of our computer is 320reqs/s. In fact, the workload fluctuates at times within a day. For example, the workload of a web server usually is lower at night. Our approach has the potential to eliminate memory leakage in child processes of the web server without any rejected requests. More specifically, we can kill most of the child process (keeping a few to respond to requests) by adjusting MaxClients to a very low value and restarting them in a clean state.

To sum up, the advantages of our approach are described as follows:

(1) Slightly adjusting the parameters of Apache can mitigate the effect of software aging and will not affect the performance of existing services or applications on the same computer system;

(2) Our approach only affects the capacity of the web server. This will not inevitably incur request loss since real-world workload is lower than the capacity of the web server at most times;

(3) Since there is sufficient time to incur request rejection, our controller will have many chances to proactively eliminate memory leakage when the real-world workload is low.

7 Conclusion

Insufficient available system resources can be one of the main causes of software aging. Previous rejuvenation techniques typically incur some system down time, which is not allowed in some safety-critical systems. This paper proposes a lightweight rejuvenation technique, which can delay system resource exhaustion as a result of software aging. This technique is implemented by adjusting the parameters of the target software system. First, we conducted a controlled experiment to explore the relationship between software aging (in terms of resource usage) and the related parameter settings. Second, a system identification method is used to extract a control model from the mass amounts of collected data. Finally, a PI controller is designed through a pole assignment method to track the resource threshold which is set by the administrator. In addition, the fastness and accuracy are evaluated by a simulation experiment.

This is the first paper explicitly describing the fundamental steps for introducing control theory into software aging. This includes the definition of input/output, control model building, control design and performance testing. This control is designed in consideration of the little inertia of a software system. In addition, our approach can be used to control other target software systems.

In the future, we will incorporate more fine-grained rejuvenation techniques such as differentiated service in our approach. In addition, since our approach rejects some requests from clients, we will evaluate the overall performance and optimal rejuvenation policy in a cluster system or virtualized system.

Acknowledgements

We are grateful to Matt Witheridge of the University of Wollongong for his valuable comments on this manuscript. This work was supported in part by the Open Fund of Tianjin Key Lab for Advanced Signal Processing (Grant No. 2017ASP-TJ04), the Natural Science Foundation of Tianjin (Grant No. 19JCYBJC15900), and the Civil Aviation Security Capacity Building Foundation of China (No. 20600605). Z.Q. Zhou acknowledges the support of an Australian Research Council linkage grant (Project ID: LP160101691).

References

[1] A. Vaidyanathan, K. S. Trivedi, A Comprehensive Model for Software Rejuvenation, *IEEE Transaction on Dependable and Secure Computing*, Vol. 2, No. 2, pp. 124-137, June, 2005.

[2] L. Kumar, A. Sureka, Feature Selection Techniques to Counter Class Imbalance Problem for Aging Related Bug Prediction, *Proceedings of 11th Innovations in Software Engineering Conference*, Hyderabad, India, 2018, pp. 1-11.

[3] J. Alonso, R. Matias, E. Vicente, A. Maria, K. S. Trivedi, A Comparative Experimental Study of Software Rejuvenation Overhead, *Performance Evaluation*, Vol. 70, No. 3, pp. 231-250, September, 2013.

[4] G. Ning, J. Zhao, Y. Lou, J. Alonso, R. Matias, K. S. Trivedi, B.-B. Yin, K.-Y. Cai, Optimization of Two-Granularity Software Rejuvenation Policy Based on the Markov Regenerative Process, *IEEE Transactions on Reliability*, Vol. 66, No. 4, pp. 1630-1646, June, 2016.

[5] H. Meng, J. J. Liu, X. H. Hei, Modeling and Optimizing Periodically Inspected Software Rejuvenation Policy Based on Geometric Sequences, *Reliability Engineering and System Safety*, Vol. 133, No. 133, pp. 184-191, September, 2015.

[6] F. Machida, J. W. Xiang, K. Tadano, Y. Maeno, Lifetime Extension of Software Execution Subject to Aging, *IEEE Transactions on Reliability*, Vol. 66, No. 1, pp. 123-134, October, 2017.

[7] W. Xie, Y. G. Hong, K. S. Trivedi, Analysis of A Two-Level Software Rejuvenation Policy, *Reliability Engineering and System Safety*, Vol. 87, No. 1, pp. 13-22, June, 2005.

[8] K. Kourai, H. Ooba, Zero-copy Migration for Lightweight Software Rejuvenation of Virtualized Systems, *Proceedings of the 6th Asia-Pacific Workshop on Systems*, Tokyo, Japan, 2015, pp. 1-8.

[9] Y. Q. Yan, P. Guo, B. Chen, Z. G. Zheng, An Experimental Case Study on the Relationship between Workload and Resource Consumption in A Commercial Web Server, *Journal of Computational Science*, Vol. 25, pp. 183-192, May, 2018.

[10] Y. F. Wang, X. R. Wang, M. Chen, X. Y. Zhu, PARTIC: Power-Aware Response Time Control for Virtualized Web Servers, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 22, No. 2, pp. 323-336, April, 2011.

[11] Y. Lu, A. Saxena, T. F. Abdelzaher, Differentiated Caching Services: A Control-theoretical Approach, *Proceedings of the 21st International Conference on Distributed Computing System*, Mesa, AZ, USA, 2011, pp. 615-622.

[12] R. Achar, P. S. Thilagam, Applications Nature Aware Virtual Machine Provisioning in Cloud, *International Journal of Ad Hoc and Ubiquitous Computing*, Vol. 27, No. 2, pp. 93-107, January, 2018.

[13] G. B. H. Bindu, K. Ramani, C. S. Bindu, Energy Aware Multi Objective Genetic Algorithm for Task Scheduling in Cloud Computing, *International Journal of Internet Protocol Technology*, Vol. 11, No. 4, pp. 242-249, September, 2018.

[14] H. Y. Cui, Y. Li, X. F. Liu, N. Ansari, Y. J. Liu, Cloud Service Reliability Modelling and Optimal Task Scheduling, *IET Communications*, Vol. 11, No. 2, pp. 161-167, January, 2017.

[15] Apache HTTP Server Version 2.0, http://httpd.apache.org/docs/2.0/mod/mpm_common.html.

[16] S. Russo, The Dual Nature of Software Aging: Twenty Years of Software Aging Research, *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Naples, Italy, 2014, pp. 431-432.

[17] C. Peng, M. Liu, X.-P. Yuan, L.-X. Zhang, J.-F. Man, A New Method for Abnormal Behavior Propagation in Networked Software, *Journal of Internet Technology*, Vol. 19, No. 2, pp. 489-498, March, 2018.

[18] J. Zhao, K. S. Trivedi, M. Grottke, J. Alonso, Y. B. Wang, Ensuring the Performance of Apache HTTP Server Affected by Aging, *IEEE Transaction on Dependable and Secure Computing*, Vol. 11, No. 2, pp. 130-141, September, 2013.

[19] K. Kourai, S. Chiba, Fast Software Rejuvenation of Virtual Machine Monitors, *IEEE Transactions on Dependable and Secure Computing*, Vol. 8, No. 6, pp. 839-851, May, 2011.

[20] P. Key, L. Massoulié, J. K. Shapiro, Service Differentiation for Delay-sensitive Applications: An Optimisation-based Approach, *Performance Evaluation*, Vol. 49, No. 1, pp. 471-489, January, 2002.

[21] Y. Abdelmalek, A. Abdelal, T. Saadawi, Media-Aware Caching Mechanism in DiffServ Networks, *Proceedings of the 6th IEEE Consumer Communications and Networking*

Conference, Las Vegas, Nevada, 2009, pp. 1-6.

[22] M. Grottke, L. Li, K. Vaidyanathan, K. S. Trivedi, Analysis of Software Aging in a Web Server, *IEEE Transaction on Reliability*, Vol. 55, No. 3, pp. 411-420, September, 2006.

[23] Y.-F. Jia, L. Zhao, K.-Y. Cai, A Nonlinear Approach to Modeling of Software Aging, *Proceedings of the 15th Asia-Pacific Software Engineering Conference*, Beijing, China, 2008, pp. 77-84.

[24] D. Mosberger, T. Jin, httperf: A Tool for Measuring Web Server Performance, *Proceedings of the First Workshop on Internet Server Performance*, Madison, Wisconsin, USA., 1998, pp. 59-67.

[25] The Linux Kernel Documentation, https://www.kernel.org/doc/html/latest/.

[26] K. J. Åstrom, B. Wittenmark, *Computer Controlled System–Theory and Design*, Englewood, Cliffs, 1984.

[27] P. Zhu, Y. M. Guo, F. Lombardi, J. Han, Approximate Reliability of Multi-state Two-terminal Networks by Stochastic Analysis, *IET Networks*, Vol. 6, No. 5, pp. 116-124, September, 2017.

Biographies

Yun-Fei Jia is currently an associate professor at Civil Aviation University of China. He received a B.E. (2001) and a M.S. (2004) degrees from HeBei University of Technology, and completed a Ph.D. in software testing at BeiHang University in 2010. His research interests include software testing and software reliability modelling.

Zhi Quan Zhou received the B.S.c degree in computer science from Peking University, China, and the Ph.D. degree in software engineering from The University of Hong Kong. He is currently an associate professor at the University of Wollongong, Australia. His current research interests include software testing and analysis.

Ren-Biao Wu was born in Wuhan, China, in 1966. He received a M.S. (1991) from Northwestern Polytechnical University and completed Ph.D. in signal processing at Xidian University. Currently, he is a professor at Civil Aviation University of China. His interests are signal processing and image processing.