CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **27**:2982–2998
Published online 17 December 2013 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3190

SPECIAL ISSUE PAPER

A lightweight software fault-tolerance system in the cloud environment

Gang Chen[1], Hai Jin[1], Deqing Zou[1,*,†], Bing Bing Zhou[2] and Weizhong Qiang[1]

[1]*Cluster and Grid Computing Lab, Services Computing Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China*
[2]*School of Information Technologies, University of Sydney, NSW 2006, Australia*

SUMMARY

With the development of cloud computing, the demand of high availability for services is growing. Unfortunately, software failures greatly reduce system availability. This paper presents a lightweight software fault-tolerance system, called SHelp, which can effectively recover programs from many types of software bugs in the cloud environment. With error virtualization techniques, it proposes 'weighted' rescue points techniques to effectively survive software failures through bypassing the faulty path. For multiple application instances running on different virtual machine, a three-level storage hierarchy with several comprehensive cache updating algorithms for rescue points management is adopted to share error handling information. On the one hand, SHelp can reduce the redundancy for multiple application instances; on the other hand, it can more effectively and quickly recover from faults caused by the same bugs. A Linux prototype is implemented on an open-source virtual machine monitor platform, Xen, and evaluated using four Web server applications that contain various types of bugs. The experimental results show that SHelp can recover server applications from these bugs in just a few seconds with modest performance overhead.

Received 24 January 2013; Revised 31 October 2013; Accepted 9 November 2013

KEY WORDS: Software Reliability; Software Self-healing; Cloud Computing; Virtual Machine; Dynamic Instrumentation

1. INTRODUCTION

Software vulnerabilities severely affect system security and availability. It can be used to attack the server applications, causing software failures and so stopping the server from providing normal services to clients. According to a survey conducted by IT industry analyst firms [1], the average business loss of an hour of IT system downtime is between $84,000 and $108,000. However, the average time needed to diagnose and generate patches is 28 days [2]. Therefore, it is a significant issue to preserve system availability, especially for server applications such as online transaction monitoring and cloud computing services.

With the rise of cloud computing, companies may purchase computing and storage capability from cloud providers, such as Amazon EC2 [3] and then deploy the application instances in the purchased resources to provide services, such as Web services, to users. It is expected that many instances of the same application can run on different virtual machines (VMs) in the same cloud environment. For example, company A could deploy Apache HTTPd application instance and MySQL application instance, and company B could deploy Apache HTTPd application instance

*Correspondence to: Deqing Zou, Cluster and Grid Computing Lab, Services Computing Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.
†E-mail: deqingzou@hust.edu.cn.

and Oracle application instance. As there are two HTTPd instances existing in the cloud, it is ideal if the error handling information can be shared between these instances, that is, the error handling information from one application instance can be used for the other application instance to quickly recover from the same faults.

Although a number of approaches help programs survive failures for traditional computing architectures, they suffer one or more of the following shortcomings: inability to prevent deterministic faults [4], a narrow suitability for only a small number of applications [5] or memory-related bugs [6], high time and space overhead [6], and incompatible with the existing programs [4]. ASSURE [7] was proposed to better address these problems by introducing rescue points and error virtualization techniques. Error virtualization is a technique used to force a heuristic-based error return in a function and rescue points are locations in the existing application code for handling programmer-anticipated faults. Once a rescue point can successfully survive the fault, ASSURE deploys and inserts codes at the appropriate rescue point to checkpoint the application state once the function is called. Thus, the server application can recover quickly from the same faults in the future.

However, there is a potential problem in ASSURE. If the appropriate rescue point is in the main procedure, the appropriate rescue point can be called very frequently during normal executions, which may cause high overhead in both space and time, and sometimes even make applications not function properly. Although ASSURE can alleviate the problem by strengthening patch test phase, it may face an even worse situation in which ASSURE may be unable to find an appropriate rescue point because selecting an upper level function as the appropriate rescue point may cause the application to exit directly in responding to a fault.

In this paper, we propose a system called SHelp for automatic self-healing for multiple application instances in a cloud environment. SHelp extends ASSURE to the cloud computing environment with two distinctive features. First, SHelp adopts a three-level storage hierarchy with several comprehensive cache updating algorithms for rescue point management. It deploys a global rescue point database in a backend console node and a two-level rescue point caches in each node. One rescue point cache is deployed in a privileged domain (called Dom0 in Xen [8]) to store all the rescue points relevant to the applications running in the physical node, and the other is in each guest operating system (called DomU) to store a small number of rescue points relevant only to applications running in the VM. With this three-level storage hierarchy for rescue points management, SHelp can dramatically reduce the redundancy because there is no need to build the same rescue point database in each VM, which makes it very convenient for multiple application instances running in different VM to contribute and share the fault-related information and thus enable more effective recovery from failures.

As mentioned earlier, ASSURE may face a potential problem when the appropriate rescue point is in the main procedure of the program. The second distinctive feature of SHelp is the introduction of 'weighted' rescue points. When an appropriate rescue point is chosen, its associated weight value is incremented. After a fault is detected, SHelp will first select the rescue point with the largest weight value to test. The rational is, if a fault occurs more frequently, the associated weight value of the appropriate rescue point becomes larger, and thus, using weight values to help selecting the appropriate rescue point may make recovery from frequently occurred faults more quickly and effectively. With the three-level storage hierarchy, weight values from multiple instances of the same application in the cloud will be added together and stored in the global rescue point database. This accumulative effect can make the same application instances quickly recover from the faults, and it also provides a useful guideline for diagnosis of serious bugs.

We have implemented a Linux prototype and evaluated it with four Web server applications that contain a wide range of bugs, including heap overflow, double-free, stack smashing, null dereference, divide-by-zero, and off-by-one. The experimental results show that SHelp can assist server applications to recover from these bugs in just a few seconds with modest performance overhead. SHelp has the following advantages:

(i) *Effective recovery*: With the weighted rescue point strategy, SHelp can avoid frequently saving the application state, and thus, it can effectively recover from the fault when ASSURE may

DOI: 10.1002/cpe

not find the appropriate rescue point or cause the application to terminate. Therefore, SHelp can be more effective in responding to users requests.

(ii) *Informative for debugging*: If a fault in a function occurred more frequently than others, the associated weight value will become very large. By checking the weight values, we are able to know where and how many times a particular fault occurred. This information is particularly useful for debugging.

(ii) *Useful error handling information sharing*: When there are a number of instances of the same application running in different VMs, SHelp can accumulate the corresponding weight values from all running instances, and thus, the accumulated weight value for the same fault may increase more quickly. This accumulative effect can help the same application instances collaboratively defense attacks, and it also greatly enhances the effectiveness in quick recovery and fault discovery.

The rest of the paper is organized as follow: The design of SHelp is described in Section 2; Section 3 discusses an implementation of SHelp; the experimental and analytical results are presented in Section 4; Section 5 gives an overview of related work; Finally, we summarize our contributions and discuss the future plan in Section 6.

2. SHELP DESIGN

In this section, we first briefly discuss the background of ASSURE and then introduce SHelp architecture, which extends ASSURE to the cloud computing environment.

2.1. Background of ASSURE

ASSURE introduces rescue points that are locations in existing application codes for handling programmer-anticipated failures. It can be used to bypass the path that induces software failures and recover from software failures by using the error virtualization technique to force an error return using an observed value in a function. ASSURE uses a production system and a triage system to implement rescue points and error virtualization. It dynamically inserts monitor codes in the entry points of the applications functions using a dynamic instrumentation tool Dyninst [8] and discovers rescue points from stress tests before the deployment of the application. The rescue points are then used to build up a *rescue-trace* graph. When the application is deployed, ASSURE uses the standard operating system error handling mechanisms to detect faults and checkpoints the application state periodically. Once detecting a fault, ASSURE transfers the latest checkpoint along with the event log to a triage system that is a shadow deployment of the application. In the triage system, ASSURE reproduces the fault according to the checkpoint and the log. After the fault is detected, ASSURE examines the call-stack and finds the matched candidate rescue points from the *rescue-trace* graph. Then, it tests the candidate rescue points in the light of the shortest distance to the faulty function one by one. If the rescue points can make the application survive the fault, it applies the tests to verify whether the candidate rescue points satisfy survivability, correctness,F and performance. After passing the tests, ASSURE creates a patch according to the appropriate rescue point and deploys the patch into the application in the production system. The patched application will trigger a checkpoint once the appropriate rescue point is called. In this way, it can roll the application back to the checkpoint to do quick recovery after detecting the same fault in the future.

We use a real-world server application Light-HTTPd [9] as an example to demonstrate how ASSURE is able to recover from buffer overflow problems. The relevant codes of Light-HTTPd are presented in Figure 1. In this example, two places may potentially cause buffer overflows. The first place (Buffer Overflow A) is in function `Log`. The solid lines with arrows on the right side of the codes and the corresponding bold codes show how the buffer overflow may occur. Assume that a user sends a GET request that contains more than 1024 bytes. When the request pointed by pointer `ptr` is passed to the function `Log` and the statement online 223 is executed to copy the data to `temp` without bounds checking, a buffer overflow is triggered because the size of array temp is only of 200 bytes. The second place causing buffer overflows for the same example is in

DOI: 10.1002/cpe

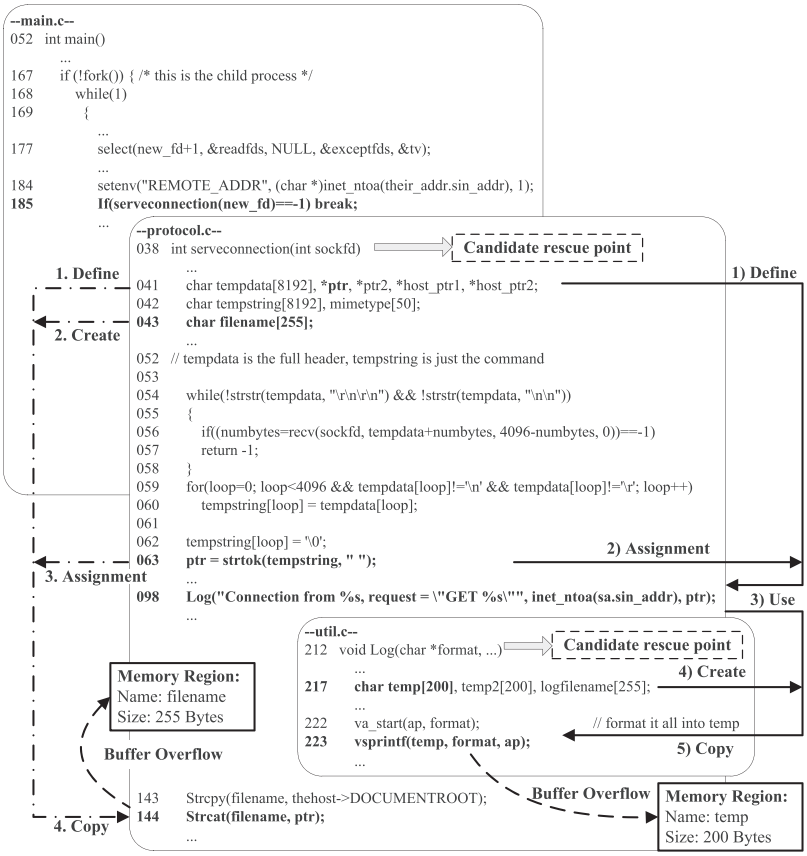

Figure 1: A graphical representation of the vulnerability in Light-HTTPd.

function `serveconnection` (Buffer Overflow B). As indicated by dashed arrow lines and the corresponding bold codes, a buffer overflow will occur for the same example.

When a fault is encountered, ASSURE reproduces the fault in the triage system, and selects candidate rescue points by matching the call-stack with the *rescue-trace* graph, that is, rescue points A and B at the entries of functions `Log` and `serveconnection` in this specific example. The application state is rolled back to a previous checkpoint, and an error value is inserted at the entry of the function to force a return when the application is replayed. This process continues until the 'faulty' function is discovered. Through bypassing the faulty function, the application is recovered from the buffer overflow without being terminated or crashed. After that, ASSURE uses tests to confirm the deployment satisfaction. If the tests are passed, ASSURE creates a patch that associates the rescue point with an error return for error virtualization and deploys the patch at the appropriate rescue point (rescue point B) in the production system. The patched application will trigger a checkpoint each time the appropriate rescue point is called during the execution. When the same fault occurs again, the application can quickly go back to the checkpoint, do the quick recovery, and continue to serve subsequent requests that may not cause the buffer overflow problem.

2.2. *Weighted rescue points*

Although ASSURE adopts tests to verify whether the rescue point meets the deployment requirements, that is, survivability, correctness, and performance, it may be unable to tolerant a fault or may cause the patched application not to function properly if an appropriate rescue points is in a loop or in the main procedure. ASSURE can use the performance evaluation to avoid deploying the rescue points, which are in loops, but when the rescue points are in the main procedure, it may face a hard choice. If the performance evaluation is strict in the test phase, the rescue points in the

DOI: 10.1002/cpe

main procedure may have not been deployed to prevent the application state from being saved too frequently, which imposes high overhead. In this case, no appropriate rescue points can be found, because if an appropriate rescue point is chosen just in an upper level function in the main procedure to avoid the high overhead, the patched application may exit directly in responding to the fault. On the other hand, if the performance evaluation is not strict in the test phase and the rescue point is deployed in the production system, it may suffer from saving the application state too frequently at that rescue point when no faults occur.

Again we take the Light-HTTPd as an example. The time ASSURE takes for a checkpoint is between 10 and 20 ms [7]. Assume that the rescue point B is deployed. When the server is heavily loaded, for example, at a rate of over 100 requests per second, the time taken just for saving the state at rescue point B will be over 1 s (assuming 10 ms for each checkpoint), and then, the server is unable to keep up with the speed of the request arrival. As a consequence, ASSURE makes the application just busy to checkpoint the application state, instead of serving users normal requests. The upper level function on the function `serveconnection` is the function `main`. If it is not a rescue point, ASSURE cannot find an appropriate rescue point to deploy. It loses effectiveness in this case.

To avoid this problem, we introduce a new strategy called weighted rescue point. In this strategy, each rescue point is assigned a weight value, which is initially set to zero. Instead of setting a checkpoint at the functions entry, the associated weight value is incremented each time a fault is found in a function. When a fault occurs, the application is rolled back to a latest checkpoint, and first uses error virtualization at a rescue point that has the largest weight value among the candidate rescue points, then at the rescue point with the second largest weight value and so on until the fault is bypassed.

It should be noted, though the number of times the application state is saved are small using our new strategy, it may take a longer time to roll the application back to a rescue point compared with the original strategy. However, the actual time taken for finding a fault depends mainly on the time for finding the rescue point where the fault occurs, but not just on the rollback time.

2.3. Application scenario

Cloud-based infrastructure is a good choice to host a variety of applications ranging from high availability enterprise services to batch oriented scientific computations. Figure 2 shows an application scenario for SHelp in cloud computing environment. As shown in Figure 2, there are several instances of the same application running in the cloud. (In this paper, the same application instances

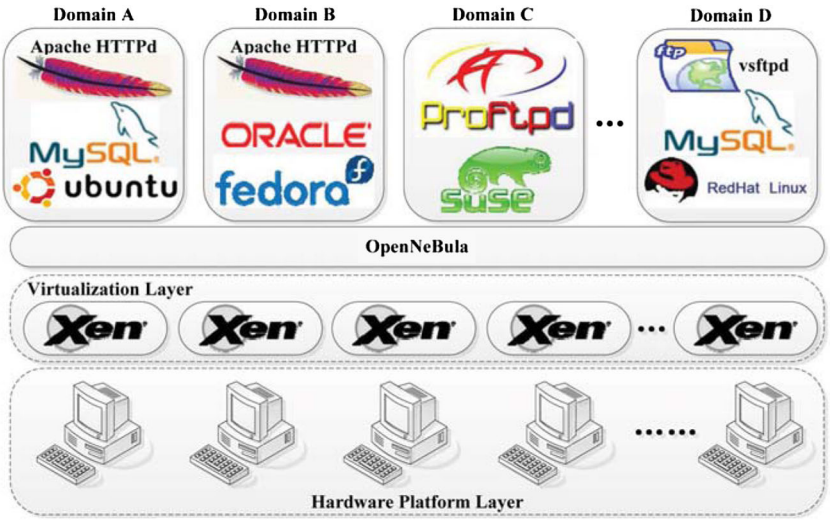

Figure 2: An application scene for SHelp.

DOI: 10.1002/cpe

mean applications are of the same version, and we dont consider the case that different configurations may cause different faults.) SHelp can share the fault-related information among DomUs, and thus, it can make the same application instances quickly recover from the same faults. For example, user A deploys Apache HTTPd application instance and MySQL application instance in Domain A, and user B deploys Apache HTTPd application instance and Oracle application instance in Domain B. Therefore, the same Apache HTTPd application instances can share the error handling information to collaboratively defense attacks. Moreover, the accumulative effect for the same application instances can also greatly enhance fault discovery.

2.4. SHelp architecture

SHelp extends ASSURE to the cloud computing environment, which is set up by OpenNebula [10]. It adopts a three-level storage hierarchy to manage rescue points and the associated weight values in a centralized manner to accelerate the fault discovery process. In order to save resources and simplify the whole recovery process, SHelp just adopts the test phase in the production system. The SHelp architecture is shown in Figure 3.

In SHelp, the ASSURE system is deployed in each VM or DomU. The ASSURE components in each host node include the following: (1) a set of sensors for detecting and identifying software faults at runtime; (2) a checkpoint and rollback component for saving the application state and rolling the application back to a previous checkpoint when a fault occurs; and (3) a test component for choosing the candidate rescue points, testing the rescue points to find where the fault occurred, and then incrementing the corresponding weight value (instead of setting a checkpoint). There is also a report component added to Dom0 for collecting the fault-related information and sending it to the frontend node, which logs and provides useful information to programmers for postmortem bug analysis.

The three-level storage hierarchy for collectively managing rescue points in SHelp has a global rescue point database. This database, residing in the frontend node, stores all applications rescue points and the associated weight values. For each host node, we adopt a two-level rescue point storage mechanism. In Dom0, there is a local rescue point database component, which is used to store the deployed applications rescue points and the associated weight values. Each DomU maintains

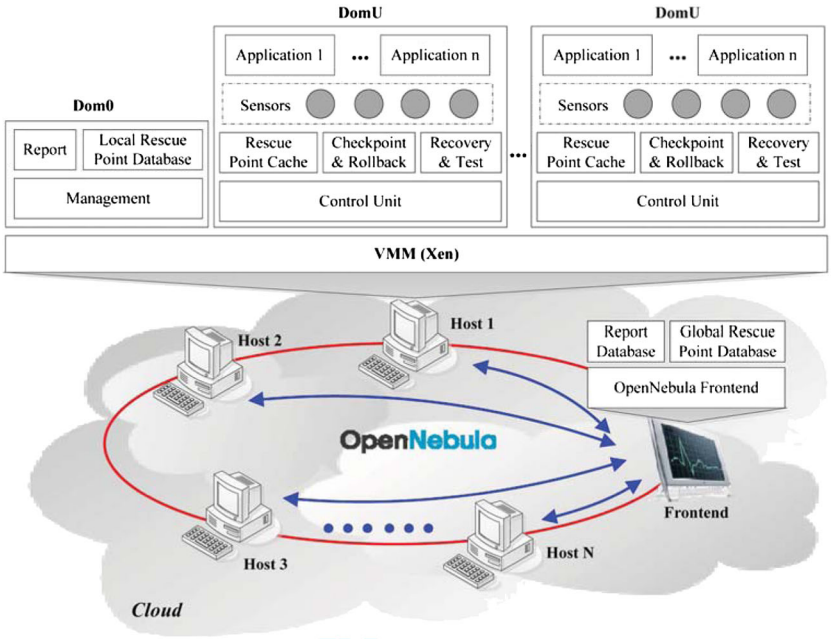

Figure 3: SHelp architecture.

DOI: 10.1002/cpe

a rescue point cache component to store a subset of relevant rescue points. To maintain the cache consistency, each VM has a control unit used to send/receive requests to/from Dom0 and to update the contents in rescue point cache, while Dom0 has a Management component used to communicate with DomU and the frontend node and to update the contents in the local rescue point database.

2.5. Fault recovery procedure in SHelp

Because SHelp adopts the concept of weight values and uses a three-level storage structure for collectively managing rescue points and weight values, it has certain differences from the original ASSURE in fault recovery. The fault recovery procedure used in SHelp is graphically depicted in Figure 4, and its main steps are discussed as follows:

Step 1: When the sensors detect a fault, the application rollbacks to the latest checkpoint. The fault-related information is analyzed, the call-stack is searched, and the candidate rescue points are determined.
Step 2: The rescue point cache is searched for information on the candidate rescue points. If the candidate rescue points are not in the cache, a request is sent to the management component in Dom0. The management component searches for the candidate rescue points from the local rescue point database. If not found, the management component sends a search request to the frontend node. After finding the candidate rescue points, the management component sends the results back, and then, the rescue point cache in DomU is updated.
Step 3: According to the weight values, the candidate rescue points are prioritized and tested one by one. The rescue point with the highest priority (or the largest weight value) will be tested first by injecting the error virtualization code at it to bypass the corresponding function when the application is replayed.
Step 4: If the fault disappears after bypassing the function, the corresponding weight value is incremented. The weight value is updated in both local rescue point cache and the local database in Dom0.
Step 5: The bug-rescue list (to be described in Section 3.5) is updated to recover the application more quickly from the same stack smashing faults.
Step 6: The weight values of rescue points are updated periodically between the local rescue point databases and the global rescue point database.
Step 7: A bug report according to the fault information and the error handling information is generated and sent to the report database component and saved in the frontend node.

3. SHELP IMPLEMENTATION

In this section, we discuss the data structure of the rescue points, the procedure on updating the rescue points, and the associated weight values, the checkpoint/rollback mechanism, and the bug-rescue list.

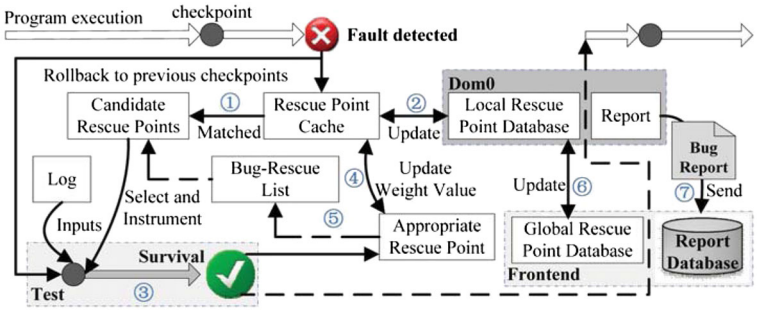

Figure 4: SHelp procedure.

DOI: 10.1002/cpe

3.1. Data structure of the rescue points

As discussed previously, SHelp stores all applications rescue points in a global rescue point database in the frontend node, the deployed applications rescued points in a local rescue point database in Dom0, and parts of applications rescue points in the rescue point cache in DomU. The data structure of the rescue points in the global rescue point database is illustrated in Figure 5, which is the same with the local rescue point database, and there are three tables in the database to store the applications rescue points.

Firstly, we need a table to distinguish applications, so SHelp uses *App_trace_table* to store the basic information of each application. In this table, the trace numbers, that is, the indexes of the sequences of function calls which are used to match the call-stack at the time when a fault occurs, are stored. It also stores a flag to indicate whether the weight values in any trace of the application have changed or not. Secondly, we need a table to store information on each trace, a sequence of function calls. Instead of storing the names of the functions, SHelp uses *App_rp_table* to store the indexes of rescue points in those function calls. In this table, a flag is used to indicate whether the weight values of the associated rescue points have changed or not in each particular trace. Lastly, we need a table to store the basic information on the rescue points for error virtualization. A table, namely *App_wrp_table*, is used to store the rescue points return values, return types, weight values, and previous weight values that is used for updating between the global and local rescue point databases.

The data structure of the rescue points in the rescue point cache is similar to that in the rescue point database. The differences are that there are no elements to store the previous weight values in the *App_wrp_table* and the flags in two tables are used to store the information to update rescue points according to the updating strategies.

3.2. Updating the rescue point cache

When the rescue point cache is full, SHelp has to unload certain rescue points before it can load a new one. Ideally, we want to keep rescue points that have a greater opportunity to be used in the near future to quickly recover from the faults. SHelp adopts a two-level updating strategy. Because SHelp uses the inter-domain communication mechanism to transfer the updating information between Dom0 and DomU, we can assume that the channel between domains is fault free.

For replacement at the application level, SHelp adopts the following updating algorithm:

Least recently used (LRU): The rescue point cache replaces the application in which the rescue points are the LRU. The element *AppFlag* in *App_trace_table* stores the time when a weight value in the application is changed. When a replacement is to take place, the element *AppFlag* in *App_trace_table* will first be checked. Although unloading an application table needs more time to be rebuilt again if there is a fault occurs in that application in the future, it can release a great amount of space in the cache.

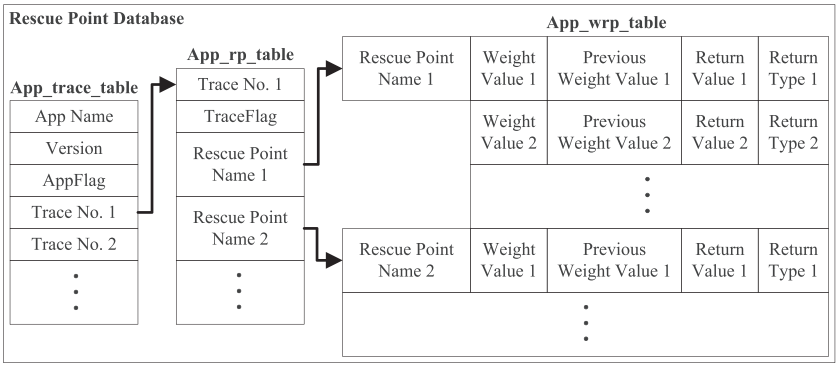

Figure 5: Data structure of the rescue points in the rescue point database.

DOI: 10.1002/cpe

For replacement at the trace level of applications, SHelp provides the following three updating algorithms:

Least frequently used weighted by the maximum weight value (LFUM): The element *TraceFlag* in *App_rp_table* stores a number which is calculated as follows:

$$TraceFlag(i) = k \bullet w_{max} + h(i) \tag{1}$$

where $h(i)$ denotes the local hit rate in a VM for trace i and w_{max} is the globally maximum weight value for trace i in this physical machine. The factor k is a tunable parameter. If k is small, the local hit rate becomes a dominant factor, and thus, the local information is counted more. If k is relatively large, the globally maximum weight value will become a dominant factor. Thus, the frequency of the fault hits in one application instance can be reflected in rescue point caches of other VMs. An appropriate value for factor k can balance the local and global situations.

LFU weighted by the average weight value (LFUA): The element *TraceFlag* stores a number that is calculated in a similar manner to LFUM, but the maximum weight value is replaced with an average weight value. In this case, it reflects the average opportunity for surviving a fault for every rescue point in a trace. The average weight values of rescue points in a trace can be calculated from all the rescue points. SHelp can also use non-zero weight values to compute the average weight value, which reflects the effectiveness of the appropriate rescue points.

Least recently/frequently used weighted by the maximum weight value (LRFUM): The element *TraceFlag* stores a number which is calculated as follows:

$$TraceFlag(i)_{init} = (k+1) \bullet w_{max} \tag{2}$$

$$TraceFlag(i) = \begin{cases} 1 + 2^{-k} TraceFlag(i) & \text{if trace } i \text{ was used at time } t, \\ 2^{-k} TraceFlag(i) & \text{otherwise.} \end{cases} \tag{3}$$

Similar to LFUM, the factor k is a tunable parameter. If the factor k goes to 0, $TraceFlag(i)_{init}$ is the maximum weight value for trace i, and $TraceFlag(i)$ is calculated as LFU, and so LRFUM becomes LFUM. In the contrast, if the factor k goes to 1, $TraceFlag(i)$ is calculated as LRU with the exponential growth. The maximum weight value can also be replaced with the average weight value which is similar to LUFA.

3.3. Updating weighted rescue points

When an appropriate rescue point is used to recover the application from a fault successfully, SHelp increases the weight value associated with the appropriate rescue points return value in the rescue point cache. At the same time, the control unit component sends the weight value updating request to the management component. Once receiving an updating request, the management component increases the weight value of the appropriate rescue point in the local rescue point database.

In addition to this real-time updating procedure, there are two periodic updating procedures among the rescue point cache, the local and global rescue point databases. In one periodic updating procedure, the management component sends updating requests to control unit component in each DomU periodically. Once receiving an updating request, the control unit component sends its *App_rp_table* to Dom0. The management component then examines the element *TraceFlag* of the traces in the local rescue point database. If there is a change in a trace, the management component will send weight values of the rescue points in this trace to the control unit component for updating. In the other updating procedure, the frontend node periodically collects the information on the changed weight values of rescue points, updates the corresponding weight values, and sends updating request to each related host. After the management component receives an updating request, it checks whether the weight values of the associated rescue points have been changed or not after the management component sends the updating information to the frontend node. If changed, an incremental is added to the corresponding weight values. At the same time, the previous weight values are set to the updating values, which are from the updating request of the frontend node.

As for the OpenNebulas schedule, one VM may be migrated from one host to another. In this case, the rescue point cache will also be migrated into the new host. In addition, the rescue points of

DOI: 10.1002/cpe

the related applications in the local rescue point database in the original host should also be moved into the new hosts local rescue point database. Moreover, if there are rescue points of the same applications in the new hosts local rescue point database, SHelp computes the incremental weight values and adds them into the corresponding weight values. If the related applications in the migrated VM are not deployed in the other VMs in the original host, SHelp sets element *AppFlag* of the related applications to zero, which means no change and the rescue points of the related applications will be removed from the local rescue point database in the next periodical updating between the local and global rescue point databases. If the related applications in the migrated VM have been deployed in the other VMs in the original host, SHelp sets all the weight values to the previous weight values and the value of all flags to zero because the situations on the changed weight values are reflected in the new host.

3.4. Checkpoint and rollback mechanism

The checkpoint and rollback component periodically checkpoints application state and automatically rolls the server application back to a previous checkpoint once detecting a fault. It also uses a log to record the requests for replaying the inputs after rollback. SHelp uses the Berkeley Lab Checkpoint/Restart (BLCR) [11] as the checkpoint and rollback tool. However, BLCR does not support the socket checkpoint. We then modified TCP Connection Passing (TCPCP) [12] and integrated it into the BLCR. TCPCP can save the TCP state when the connection is established. It adopts function `tcpcp_getici` to get the information that it needs to restore a TCP connection endpoint in a data structure called internal connection information (ICI) that contains ipv4 addresses of the server applications and the clients, ports at the server applications and the clients, TCP flags, send windows scale, receive windows scale, maximum segment size at the server applications and the clients, TCP connection state (e.g., ESTABLISHED), sequence number of next byte to send and to receive expectedly, window received from peer and advertised to peer, and timestamps. The function `tcpcp_setici` restores the TCP connection according to ICI. If the TCP connection is not established, however, it does not work. In other words, it does not work if the load of the server application is not very high because there are idle processes or threads waiting for users to connect (i.e., the TCP connection state is not ESTABLISHED). Because of this, we extended it from the old Linux kernel 2.6.11 to Linux kernel 2.6.18.8 to satisfy the requirement of Xen 3.2.0, and modified `tcpcp_getici`, `tcpcp_setici` and other related functions to enable BLCR to utilize these two functions.

Two empty functions in BLCR can be used to save and restore the socket. According to the state of the TCP socket, we should consider two situations. One situation is that if the connection is established, BLCR can directly use the modified function `tcpcp_getici` to get ICI and save it in the checkpoint file. When rolling back, BLCR can get ICI from the checkpoint file and use the modified function `tcpcp_setici` to restore the state of the TCP socket. The other situation is that if the connection is not established, the most of the TCP socket information is those values being set initially. BLCR can only save several important arguments including the port and ipv4 address of the server applications. When rolling back, BLCR just sets the saved important arguments in the socket to wait for connecting.

3.5. Bug-rescue list

For quick recovery from a stack smashing fault, SHelp uses a bug-rescue list to record the appropriate rescue point related to the fault (step 5, as illustrated in Figure 4). According to the bug-rescue list, once a stack smashing fault is detected again, SHelp can optimistically use the appropriate rescue point to first test if the stack smashing fault occurs previously. In this way, SHelp can avoid frequently inserting monitor codes into the server application. Although SHelp should insert monitor codes to get the call-stack again once the appropriate rescue point does not handle the problem successfully in cases when there are less than one stack smashing bug occurred in the application, it can quickly survive the same stack smashing fault with a great opportunity.

DOI: 10.1002/cpe

4. EXPERIMENTAL EVALUATION

We have implemented a SHelp prototype system on Linux. It consists of user-space modules and loadable kernel modules for Linux 2.6.18.8 kernel modified with TCPCP support in Fedora 8. Dyninst 6.0 [13] is used for runtime code injection. We evaluate the effectiveness of SHelp on real-world bugs from four server applications. Our experimental platform consists of five machines and a 100 Mbps Ethernet connection among them, and they are deployed with OpenNebula 2.2 and Xen 3.2.0. The configuration of the frontend is two Intel Xeon E5405 quad core 2 GHz processors with 12 MB L2 cache, and 8 GB memory, host 1 is Intel Xeon E5300 dual core 2.6 GHz processors with 2 MB L2 cache, and 1 GB memory, host 2 is the same as the frontend, host 3 is Intel Xeon E6550 dual core 2.33 GHz processors with 4 MB L2 cache, and 2 GB memory, and the client machine has Intel E5200 dual core 2.5 GHz processors with 2 MB L2 cache, 2 GB memory. In our experiment, each DomU is allocated with two virtual CPUs and 512 MB memory. Figure 6 shows the experimental environment in which we carry out our test.

In the following, we first present the overall experimental result for SHelp. Then, we give an effective recovery analysis by comparing SHelp with ASSURE. After that, we present the recovery performance of SHelp for each bug and the benefits of the bug-rescue list. Finally, the checkpoint/rollback overhead is analyzed.

4.1. Bug list and overall effectiveness analysis

As shown in Table I, there are four Web server applications: Apache, Light-HTTPd, ATP-HTTPd [14], and Null-HTTPd (Nullhttpd) [15] used in our experiment. Three of them excluding Apache HTTPd server are lightweight Web server applications. The Web server applications contain various types of bugs, including heap overflow, double-free, stack smashing, NULL dereference, divide-by-zero, and off-by-one. We evaluate SHelp prototype using eight bugs. Among them, six bugs were introduced by the original developers. The other two, that is, double-free and divide-by-zero bugs, were injected into Null-HTTPd server and Light-HTTPd server, respectively. To simulate bug occurrences in real scenarios, we mix the normal inputs with the bug triggering inputs. Apache benchmark ab [16] is used to test the server application, and malicious requests are sent every 20 s to trigger a fault.

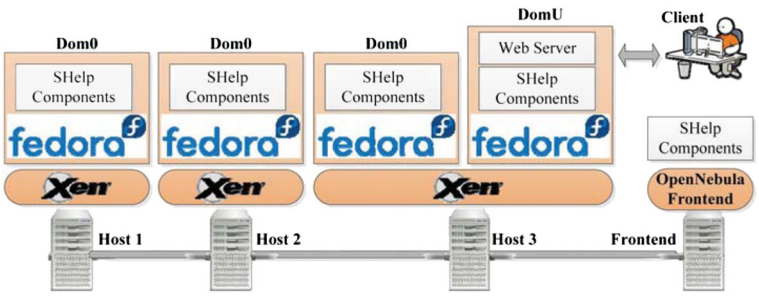

Figure 6: The experiment environment.

Table I. Applications and bugs used in evaluation.

Application	Version	Bug	Depth	Value
Apache	2.0.49	Off-by-one	2	−1
	2.0.50	Heap overflow	2	20,014
	2.0.59	NULL dereference	2	502
Light-HTTPd	0.1	Stack smashing	2	−1
Light-HTTPd-dbz		Divide-by-zero	2	−1
ATP-HTTPd	0.4b	Stack smashing	1	0
Null-HTTPd	0.5.0	Heap overflow	1	Void
Null-HTTPd-df		Double free	3	0

DOI: 10.1002/cpe

The overall effectiveness of the SHelp is illustrated in Table I. The column named *value* illustrates the rescue value, which is used to propagate errors, and the value in column *depth* indicates the rescue distance, that is, the number of rescue points between the function where the fault is detected and the function where the appropriate rescue point is placed. The shorter the rescue depth is, the more quickly the system is able to recover from a fault. The average observed rescue distance in the table for all bugs is two. This is the same as that obtained using ASSURE.

4.2. Effective recovery from the faults

We focus on extending ASSURE with weighted rescue points and fault-related information sharing in VMs. Because ASSURE is not an open source, to compare ASSURE with our SHelp, we re-implemented the main functions of the ASSURE system and also made some minor changes. Firstly, the rescue point analysis and test phase are moved from the offline triage system into the online production system. Secondly, a modified BLCR tool is used as ASSUREs checkpoint tool. Finally, as our test module just monitors whether the instrumented server applications crash or not from replaying a number of recent inputs, we make the test phase in ASSURE the same as that in SHelp.

The VM that contains the tested server applications is deployed in a lightly loaded node. In our experiment, they are deployed in host 3 that hosts only one VM. Figure 7(a) shows the results obtained from SHelp and ASSURE, respectively, on effectiveness of recovery from the faults for a Web server application Light-HTTPd. As shown in Figure 7, for stack smashing bug in the Light-HTTPd Web server application (which is illustrated in Figure 1), ASSURE may select the function `serveconnection` as the appropriate rescue point. In this case, ASSURE suffers from frequently saving the application state, which causes the server application to be very slow in responding to user normal requests, that is, the throughput is only about 253 KB/s as shown in Figure 7(a). If the function `serveconnection` is not chosen as the appropriate rescue point because it may not pass through the performance analysis in the test phase in ASSURE, ASSURE is unable to recover from the fault as it cannot find an appropriate rescue point. In contrast, SHelp can effectively recover from the fault via the weight value associated with a rescue point at the beginning of the function. Furthermore, SHelp can also avoid frequent state saving when the function `serveconnection` is selected as the appropriate rescue point.

Figure 7(b) shows the experimental results from SHelp and ASSURE on effectiveness of recovery from faults for a Web server application Null-HTTPd. For the artificially injected double free bug in Null-HTTPd used in Figure 7(b), the function call sequence is as follows: `main`, `accept_loop`, `htloop`, `dorequest`, and `read_header`. Function `read_header` is injected a double free bug and ASSURE may select function `htloop` as the appropriate rescue point. In this case, every time a users request arrives, ASSURE will take a checkpoint. It makes the application busy taking checkpoints and thus causes the server application to be very slow in responding to users requests when a large number of users requests arrive (the throughput is about 56 KB/s in our experiment). If function `htloop` is not chosen as the appropriate rescue point because it cannot pass through the performance analysis in the test phase, ASSURE should select the function `accept_loop` as

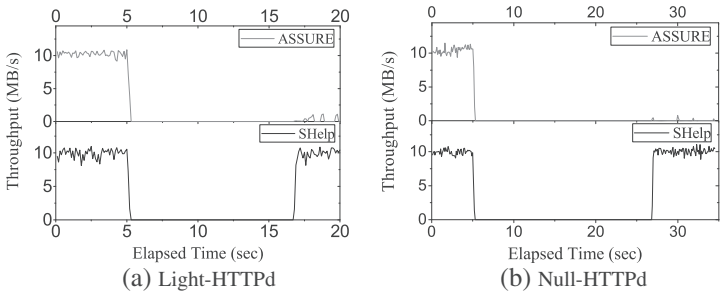

Figure 7: Comparison between ASSURE and SHelp. (a) Light-HTTPd; (b) Null-HTTPd .

DOI: 10.1002/cpe

a candidate rescue point to test for that it the only one upper level rescue point, and it will rollback many times to a distant checkpoint which may not exist because the application state may still be inside function `accept_loop`. In this case, the error virtualization cannot be adopted. Even if the application state can roll back to the position before function `accept_loop` is called, error virtualization can cause the application to exit directly. Either situation makes ASSURE not recover from faults effectively. In contrast, adopting weighted rescue points, SHelp avoids frequently saving the application state and does not need to face the tough choices.

4.3. Recovery performance

To measure the average time for fault recovery, we first consider the case when a fault was not previously detected in a local VM. When a trace path is not found in the local rescue point cache to match the call-stack, a request will be sent to Dom0 to get a set of rescue points for error virtualization. Dom0 should also look up the local rescue point database and send query request to the frontend if not found. This query time incurs extra overhead and the communication is the main overhead. However, the query overhead can be greatly reduced for inter-domain communication in the cloud computing.

If the weight values of the rescue points in the matched trace path are all zero, the fault is new to the system, and it may take a relatively long time to locate the faulty function and recover. The average time required to recover from a new fault for each of eight bugs is depicted in Figure 8 and marked as 'First-1'. The average time in the figure consists of three parts: *analysis* time is the time to rollback the application to pervious checkpoints and analyze the core dump to get the call-stack; *instrument* time is the first time spent by Dyninst to attach the Web server application and force the function at a rescue point to return an error value; and *test* time is the elapsed time for testing whether the chosen rescue point is the appropriate rescue point until the appropriate rescue point is found.

We then consider the average recovery time for cases when a fault has occurred before in a local VM. If the fault has been detected before, weight values of rescue points in a trace path are not all equal to zero. SHelp selects a rescue point with the largest weight value for the instrument function to test first, and then selects the one with the second largest weight value and so on until the fault is bypassed. Making use of weight values, the recovery time, marked 'First-2' in the figure, could be greatly reduced. (The recovery time marked First-2 is measured after a period (10 min in our experiment) when the fault first occurs in a local VM.) In this experiment, we deploy two VMs in host 3 and apply the LFUM algorithms as our update strategy of the rescue point cache.

It should be noted that, if the rescue distance (or depth) is equal to one, the appropriate rescue point is the nearest one to the 'faulty' function. In this case, the rescue point with the largest weight value maybe the nearest one with a great opportunity, and testing the nearest rescue point for the fault first time occurring is the same with testing the rescue point with largest weight value for the fault occurring again. Thus, the benefit can be obtained from using weight values of rescue points only when the rescue distance is greater than one. Our experimental results also indicate that the benefit of using weight values is more significant for deeper rescue distances. For example, the

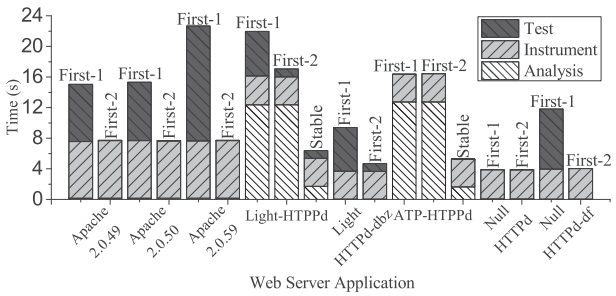

Figure 8: Average recovery time.

DOI: 10.1002/cpe

Apache Web server application with a NULL dereference bug has a rescue distance of three. The recovery time in the case when the fault was detected previously (First-2 in Figure 8) is about 2.93 times shorter than the time required for handling a new fault (First-1).

Once a stack smashing bug occurs, the overhead of getting the call-stack is very high. Therefore, SHelp uses a bug-rescue list to optimistically choose an appropriate rescue point for error virtualization when a stack smashing bug occurs. It can dramatically reduce the overhead for fault recovery with a great opportunity. As shown in Figure 8, the bar marked 'stable' denotes the time for fault recovery using the bug-rescue list in cases when the same stack overflow bugs occurs again in a local VM. In our experiment, the total recovery time can be reduced from 22.65 s (when the fault first occurs) to 3.875 s (when the same fault occurs again) for Light-HTTPd, and from 17.57 and 3.646 for ATP-HTTPd, indicating a significant improvement after the bug-rescue list is applied.

4.4. Checkpoint/rollback overhead analysis

Figure 9 shows the average time required for checkpointing and restoring the state of the Web server applications. Compared with the time for a whole Web server application reboot after a fault, the restoration time can be considered very small. In our experiment, when there is a fault occurring, the Apache 2.0.49 needs 1.1 s to reboot and restart the normal service, but the rollback time only takes 41 ms to serve future requests. In general, during the normal operation, SHelp monitors the application using the standard operating system error handling. The extra overhead incurred from adopting periodic checkpoint mechanism can be considered small for the Web server applications used in the experiments.

SHelp uses BLCR for lightweight checkpoint and rollback. Because the current BLCR does not support copy-on-write, every checkpoint has to save the full application state. The Web server application state sizes range from 265 KB for ATP-HTTPd to 20.8 MB for Apache 2.0.59. In the near future we shall implement the copy-on-write mechanism in BLCR to greatly improve the system performance.

5. RELATED WORK

There are many approaches that have been proposed to deal with software bugs. They can be classified into three categories, that is, checkpoint and rollback, bug detection, and self-healing technology.

5.1. Checkpoint and rollback

Several tools are developed to provide lightweight checkpoint and rollback mechanisms, such as Flash-Back [17] and multi-process checkpoint tool [18]. And also, there are many other tools that support heavyweight checkpoint and rollback mechanisms, such as Revirt [19] and Time-traveling Virtual Machines [20]. Revirt is a system-level checkpoint mechanism to do postmortem analysis, which is used to analyze system behavior, but its overhead is very high. In SHelp, we adopt BLCR, an open source tool that provides lightweight checkpoint and rollback mechanism. However, BLCR

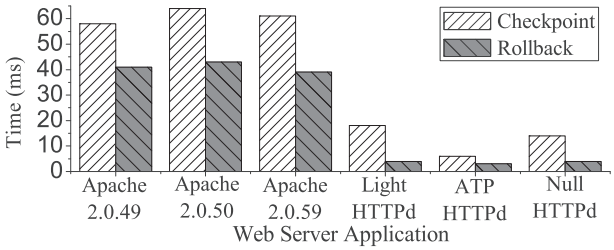

Figure 9: Checkpoint/rollback time.

DOI: 10.1002/cpe

does not support socket checkpoint. Because TCPCP allows cooperating applications to pass ownership of TCP connection endpoints from one Linux host to another, a modified TCPCP is developed for SHelp to support socket checkpoint.

5.2. Bug detection

There exist several tools for bug detection, and they have advantages and disadvantages. Among these tools, the overhead of address space randomization [21] and ASLR [22] can almost be ignored, but they can be bypassed probabilistically [23]. There are also less lightweight tools, which can provide bug protection, such as StackGuard [24] and CCured [25] that require the applications source code, while Purify [26] and Valgrind [27] can only work with binary code.

SHelp is not tied to any specific bug detection tool and uses the standard operating system error handling as the bug detection tool at present. However, it is very easy for the system to integrate more sophisticated tools and so more complex bugs can be detected. This will be our future work.

5.3. Self-healing technology

Reboot technology attempts to restart the program to deal with faults. It includes the whole program restart [28], rebooting faulty parts of software components (Micro-reboot [4]), and software rejuvenation [29], which is an active approach to reboot the program to deal with software aging. Restarting the whole program may cause a long period of downtime, especially for large server applications. To alleviate this problem, Micro-reboot is proposed to reboot only faulty parts of software components. Though application servers may recover from faults more quickly than the whole program restart, it needs the programmers to redesign the whole program. Moreover, the reboot technology cannot deal with deterministic bugs because deterministic bugs will induce the program failures again after the program is rebooted.

Rx [5] combines the checkpoint/rollback mechanism and a changing execution environment to deal with bugs. It uses proxy to make the clients unaware of failures and drop the malicious input requests to avoid attacks. However, 72–87% of the faults are independent of the operating environment, and recovering from these faults requires the use of application-specific knowledge [30]. Therefore, Rx is only suitable for a small number of applications.

Failure-oblivious computing [6] deals with memory-related bugs by modifying the memory interface to mask memory errors. It manufactures values for ‘out of the bounds read’ and discards ‘out of the bounds write’. However, it incurs high overhead (1–8× slowdowns) and may result in unpredicted behavior which imposes a new threat to the application.

Selective Transaction Emulation (STEM [31]) is similar to the failure-oblivious computing but speculatively emulates the faulty region of codes in a virtual core. To recover from a fault, it first introduces error virtualization to force a heuristic-based error value from a function where a fault occurs. However, STEM needs applications source code to recompile and restart the application to speculatively emulate a scope of instructions on the virtual processor when a fault occurs. STEM also cannot guarantee that it wont introduce a new threat.

Our previous work SafeStack [32] proposes the memory access virtualization mechanism that can dynamically move the stack objects into a protected memory area. Based on this mechanism, SafeStack can automatically patch the stack buffer overflow vulnerability. Our previous works Memshepherd [33] and OPSafe [34] can uniformly manage stack and heap objects to safely protect memory objects. All these solutions can only deal with memory-related vulnerabilities.

Virtualization technology has been used in Byzantine fault tolerance [35] to provide diverse replications in a single-machine (LBFT) [36]. LBFT uses VMs to provide diverse operating systems in a single-machine instead of using many physical machines to achieve the same goal and enhances the Byzantine fault tolerance of single-machine servers to software attacks or errors. But it needs system and application diversity and is not suitable for the situation where the resources are scarce.

DOI: 10.1002/cpe

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a lightweight runtime system SHelp that can survive software faults for server applications running in the cloud environment. SHelp can be considered as an extension of ASSURE to the cloud environment. To make the system perform more effectively and efficiently, however, we introduced two new techniques, namely, weighted rescue points and three-level storage hierarchy with several cache updating algorithms for rescue point management. With weighted rescue points, the system can effectively recover from faults, which are difficult for the original ASSURE to handle. With a three-level rescue point storage hierarchy, the error handling information can be shared among applications in different VMs, which enables applications to recover from the same faults more quickly. A prototype system is developed, and the experimental results obtained from four Web server applications demonstrate that the system is able to recover from many types of faults effectively and efficiently with only modest overhead.

In the near future, we shall implement full functions of ASSURE and integrate more sophisticated tools into SHelp, as well as test the effectiveness of our system for more complex server and client applications.

ACKNOWLEDGEMENTS

This paper is supported by National 973 Plan of China under grant No.2014CB340600, and National Science Foundation of China under grant No. 61272072. The preliminary result of this paper is published in Cluster 2010 [37].

REFERENCES

1. Vision solutions staff. assessing the financial impact of downtime. Vision Solutions, Inc.
2. Internet security threat report. http://www.symantec.com/enterprise/threatreport/index.jsp [Accessed date 2011].
3. Amazon EC2, Amazon Elastic Compute Cloud, 2011. http://aws.amazon.com/ec2/ [Accessed date 2011].
4. Candea G, Kawamoto S, Fujiki Y, Friedman G, Fox A. Microreboot - a technique for cheap recovery. *Proceedings of the 6th Symposium on Opearting Systems Design & Implementation*, 2004; 3–3.
5. Qin F, Tucek J, Sundaresan J, Zhou Y. Rx: treating bugs as allergies—a safe method to survive software failures. *Proceedings of the 20th ACM Symposium on Operating System Principles,* 2005; **39**(5):235–248.
6. Rinard M, Cadar C, Dumitran D, Roy DM, Leu T, Beebee Jr W. Enhancing server availability and security through failure-oblivious computing. *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, USENIX Association, 2004; 303–316.
7. Sidiroglou S, Laadan O, Perez C, Viennot N, Nieh J, Keromytis A. ASSURE: automatic software self-healing using rescue points. *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* 2009; **44**(3):37–48.
8. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. *Proceedings of the 19th ACM Symposium on Operating System Principles* 2003; **37**(5):164–177.
9. A light http server and content management system. http://lhttpd.sourceforge.net/ [Accessed date 2013].
10. OpenNebula project, 2011. http://www.opennebula.org/ [Accessed date 2011].
11. Hargrove P, Duell J. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Proceedings of the 2nd Scientific Discovery through Advanced Computing Program Conference*, Denver, USA, 2006; 494–499.
12. Almesberger W. TCP connection passing. *Proceedings of the 2004 Linux Symposium*, Ottawa, Ontario, Canada, 2004; 9–29.
13. Buck B, Hollingsworth J. An API for runtime code patching. *International Journal of High Performance Computing Applications* 2000; **14**(4):317–329.
14. Atphttpd remote GET request buffer overrun vulnerability. http://www.securityfocus.com/bid/8709/discuss/ [Accessed date 2013].
15. Nulllogic, the NULL HTTPD server, 2011. http://nullwebmail.sourceforge.net/httpd/ [Accessed date 2011].
16. ab, Apache HTTP server benchmarking tool, 2011. http://httpd.apache.org/docs/2.0/programs/ab.html [Accessed date 2011].
17. Srinivasan S, Kandula S, Andrews C, Zhou Y. Flashback: a light-weight rollback and deterministic replay extension for software debugging. *Proceedings of the 2005 USENIX Annual Technical Conference*, USENIX, 2004; 29–44.
18. Laadan O, Nieh J. Transparent checkpoint-restart of multiple processes on commodity operating systems. *Proceedings of the 2007 USENIX Annual Technical Conference*, USENIX, 2007; 323–336.
19. Dunlap G, King S, Cinar S, Basrai M, Chen P. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *Proceedings of the 5th Symposium on Operating System Design and Implementation* 2002; **36**(SI):211–224.

DOI: 10.1002/cpe

20. King S, Dunlap G, Chen P. Debugging operating systems with time-traveling virtual machines. *Proceedings of the 2005 USENIX Annual Technical Conference*, USENIX, 2005; 1–15.
21. Bhatkar S, DuVarney D, Sekar R. Address obfuscation: an efficient approach to combat a board range of memory error exploits. *Proceedings of the 12th USENIX Security Symposium*, USENIX, 2003; 105–120.
22. Pax Team, Address Space Layout Randomization, 2011. http://pax.grsecurity.net/docs/aslr.txt [Accessed date 2011].
23. Shacham H, Page M, Pfaff B, Goh E, Modadugu N, Boneh D. On the effectiveness of address-space randomization. *Proceedings of the 11th ACM conference on Computer and Communications Security*, ACM, 2004; 298–307.
24. Cowan C, Pu C, Maier D, Hintony H, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. *Proceedings of the 7th USENIX Security Symposium*, USENIX, 1998; 63–78.
25. Necula G, McPeak S, Weimer W. Ccured: Type-safe retrofitting of legacy code. *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 2002; 128–139.
26. Hastings R, Joyce B. Purify: fast detection of memory leaks and access errors. *Proceedings of the USENIX Winter 1992 Technical Conference*, USENIX, 1991; 125–138.
27. Nethercote N, Seward J. Valgrind: a program supervision framework. *Proceedings of the 3rd International Workshop on Runtime Verification*, Boulder, Colorado, USA, 2003; 44–66.
28. Sullivan M, Chillarege R. Software defects and their impact on system availability-a study of field failures in operating systems. *Proceedings of the 21th Annual International Symposium on Fault-Tolerant Computing*, IEEE, 1991; 2–9.
29. Huang Y, Kintala C, Kolettis N, Fulton N. Software rejuvenation: analysis, module and applications. *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing*, IEEE, 1995; 381–390.
30. Chandra S. An evaluation of the recovery-related properties of software faults. *Ph.D. Thesis*, University of Michigan, 2000.
31. Sidiroglou S, Locasto M, Boyd S, Keromytis A. Building a reactive immune system for software services. *Proceedings of the 2005 USENIX Annual Technical Conference*, USENIX, 2005; 149–161.
32. Chen G, Jin H, Zou D, Zhou B, Liang Z, Zheng W, Shi X. Safestack: automatically patching stack-based buffer overflow bugs, 2013; 369–379.
33. Zou D, Zheng W, Jiang W, Jin H, Chen G. Memshepherd: comprehensive memory bug fault-tolerance system. *To appear in Security and Communication Networks* 2013.
34. Chen G, Jin H, Zou D, Dai W. On-demand proactive defense against memory vulnerabilities. In *Proceedings of the 10th IFIP International Conference on Network and Parallel Computing*. Springer: Guiyang, China, 2013; 368–379.
35. Castro M, Liskov B. Practical byzantine fault tolerance. *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, 1998; 173–186.
36. Chun B, Maniatis P, Shenker S. Diverse replication for single-machine byzantine-fault tolerance. *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, USENIX, 2008; 287–292.
37. Chen G, Jin H, Zou D, Zhou B, Qiang W, Hu G. Shelp: automatic self-healing for multiple application instances in a virtual machine environment. *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, IEEE, 2010; 97–106.

DOI: 10.1002/cpe