

What is System Hang and How to Handle it

Yian Zhu¹, Yue Li², Jingling Xue², Tian Tan³, Jialong Shi¹, Yang Shen³, Chunyan Ma³

¹*School of Computer Science, Northwestern Polytechnical University, Xi'an, P.R.China*

²*School of Computer Science and Engineering, University of New South Wales, Sydney, Australia*

³*School of Software and Microelectronics, Northwestern Polytechnical University, Xi'an, P.R.China*

{zhuya,machunyan}@nwpu.edu.cn

{yueli,jingling}@cse.unsw.edu.au

{silverbullettt,jialong.tea.yangfields}@gmail.com

Abstract

Almost every computer user has encountered an unresponsive system failure or system hang, which leaves the user no choice but to power off the computer. In this paper, the causes of such failures are analyzed in detail and one empirical hypothesis for detecting system hang is proposed. This hypothesis exploits a small set of system performance metrics provided by the OS itself, thereby avoiding modifying the OS kernel and introducing additional cost (e.g., hardware modules). Under this hypothesis, we propose SHFH, a self-healing framework to handle system hang, which can be deployed on OS dynamically. One unique feature of SHFH is that its “light-heavy” detection strategy is designed to make intelligent tradeoffs between the performance overhead and the false positive rate induced by system hang detection. Another feature is that its diagnosis-based recovery strategy offers a better granularity to recover from system hang. Our experimental results show that SHFH can cover 95.34% of system hang scenarios, with a false positive rate of 0.58% and 0.6% performance overhead, validating the effectiveness of our empirical hypothesis.

Keywords-System Hang, Operating System, Self-Healing Framework, Fault Detection and Recovery

1. Introduction

Almost every computer user has encountered such a scenario in which all windows displayed on a computer monitor become static and the whole computer system ceases to respond to user input. Sometimes even the mouse cursor does not move either. “Unresponsiveness”, “freeze” and “hang” have been used to describe such a phenomenon, with “hang” being the most popular [1]–[4], [6], [7], [9], [12]. Note that a single program unresponsive failure (i.e., one application failing to respond to user input) is regarded as *application hang*, which is not the focus in this paper. Unlike the other failures (e.g., invalid opcode and general protection fault) whose causes can be detected directly by hardware [13], system hang cannot usually be detected by hardware or even perceived

by operating system (OS) (except for some severe cases detected only partially by watchdog mechanisms provided by some modern OSes). This leaves the user no choice but to power the system off. As a result, the OS fails to provide continuous services, causing the user to lose some valuable data. Worse still, if the computer system is deployed in some mission-critical applications, e.g., nuclear reactors, system hang may lead to devastating consequences.

By observing existing studies dealing with system hang, we draw two conclusions. First, most studies, although being effective in certain cases, could only address certain system hang scenarios [1]–[5]. One main explanation to this is that it is difficult to analyze the causes of system hang, and accordingly, each study focuses on its own assumptions about the causes of system hang. As a result, it is necessary to study the causes of system hang more comprehensively.

Second, most methodologies for detecting system hang need additional assistance, provided by either new hardware modules [7], modified OS kernels [1], [5], or monitor breakpoints inserted dynamically for interested code regions [4]. Can we rely on the existing services provided by the OS to detect system hang effectively? An attempt made in [2] does this by just monitoring I/O throughput, but it fails if a hang occurs within some OS code not related to I/O. The work of [8] is developed on the assumption that statistical models of processes, for such metrics as CPU and memory utilization, may reveal the slowness of the system (similar to system hang). However, since the causal relationship between the statistical models for processes and the slowness for the system has not been validated, the effectiveness of this assumption remains unclear. As a result, whether or not existing OS services can be utilized to detect system hang becomes an attractive argument, since an affirmative answer implies that no additional cost will be incurred.

The main contributions of this paper are as follows. We give a new characterization of system hang based on the two popular views about it (as described in Section 2.1). Besides, the causes of system hang are analyzed in detail from two aspects: indefinite wait for system resources (resources not released or released slowly) and infinite loop under interrupt and

preemption constraints. Accordingly, we present six types of faults responsible for system hang.

We propose a self-healing framework to handle system hang automatically and refer to it as SHFH, which can be deployed on OS (currently implemented on Linux) dynamically. One unique feature is that a “light-heavy” detection strategy is adopted to make intelligent tradeoffs between the performance overhead and the false positive rate induced by system hang detection. Another feature lies in its diagnosis-based recovery strategy, which is designed to provide a better granularity for system hang recovery.

We have selected UnixBench [22] as our benchmark suite, and injected six types of faults into UnixBench to cause system hang among 9 bench workloads representing at least 95% of kernel usage [26]. By analyzing a total of 68 performance metrics (e.g., context switches per second and number of runnable tasks) which are provided by the OS itself from 1080 experiments under normal and anomalous workloads, and after further experimental validation by using both UnixBench and LTP (Linux Test Project) [21], we find that 9 common performance metrics are sufficient as the basis to detect most system hang problems without requiring any additional assistance (e.g., new hardware modules or kernel modification).

The rest of this paper is organized as follows. Section 2 describes what system hang is and what causes it. Section 3 discusses whether empirical system performance metrics can be utilized to detect system hang. According to the hypothesis presented in Section 3, SHFH is proposed and described in detail in Section 4. Section 5 evaluates our SHFH and validates accordingly the effectiveness of the hypothesis made in Section 3. Section 6 discusses the related work and Section 7 concludes the paper.

2. System Hang and Causes

There is no standard definition of system hang. In Section 2.1, we give a new characterization of system hang as our analysis foundation according to the two existing views about it. The causes of system hang are analyzed in detail in Section 2.2.

2.1. What is System Hang

There are two popular views. Studies [1], [3], [5], [7] describe system hang as that OS does not relinquish the processor, and does not schedule any process, i.e., the system is in a totally hang state which does not allow other tasks to execute and respond to any user input. On the other side, studies [2], [4], [8], [9], [11] consider that when OS gets partially or completely stalled, and does not respond to user-space applications, the system enters a state of hang.

We prefer the second view about system hang because it includes a broader scope of hang scenarios which is in accordance with our daily human-computer interaction experience, and based on which,

a new characterization of system hang is given below.

System hang is a fuzzy concept which depends on the criteria of the observer - the system gets partially or completely stalled, and most services become unresponsive, or respond to user inputs with an obvious latency (an unacceptable length of time according to the observer).

2.2. Causes of System Hang

Tasks need to run effectively to provide services. In other words, if tasks cannot run, or run without doing useful work, users would be aware of the unavailable services (unresponsive). Accordingly, what causes tasks to be unavailable to run (i.e., tasks to wait for resources that will never be released) or to do useless work (i.e., tasks to fall into an infinite loop) contributes to system hang. It should be noticed that although a task falls into an infinite loop, it can be interrupted or preempted by other tasks. Besides, some system hangs can be automatically recovered after a period of time since the resources which are held by other tasks are released slowly. In this situation, if users have no patience to wait for a long time (until resources are released), system hang is considered happening.

Consequently, we analyze the causes of system hang from two aspects: infinite loop under interrupt and preemption constraints and indefinite wait for system resources (resources not released or released slowly). Accordingly, six types of faults are distinguished as shown in Figure 1.

2.2.1. Infinite Loop

When interrupts are disabled (F1), even a clock interrupt cannot be responded. As a result, if the running task does not relinquish the CPU on its own, i.e., falls into an infinite loop, other tasks would have no chance to be executed. In the case with interrupts enabled but preemption disabled (F3), CPU can respond to interrupts; however, even tasks with higher priority cannot be executed, thus making some services provided by the ready tasks unavailable. Although both interrupts and preemption are enabled, when a task falls into an infinite loop in kernel (F2) (certain OSes, e.g., Linux after 2.6 version, support kernel preemption mechanism), it still cannot be preempted unless all the locks held by the task are released or the task is blocked or explicitly calls schedule function; however, falling into an infinite loop in kernel offers little chances to satisfy the above conditions, thus providing OS little opportunities to schedule other tasks. Generally, infinite loops can be explained in two scenarios: (1) an interrupt (preemption) enabled operation cannot be executed due to an infinite loop formed earlier and (2) an interrupt (preemption) disabled/enabled pair falls inside an infinite loop. Faults related to spinlocks, e.g., double spinlocks, are also categorized into F1 (the first scenario) due to its mechanism of busy waiting for locks after interrupts are disabled. Even in a multi-

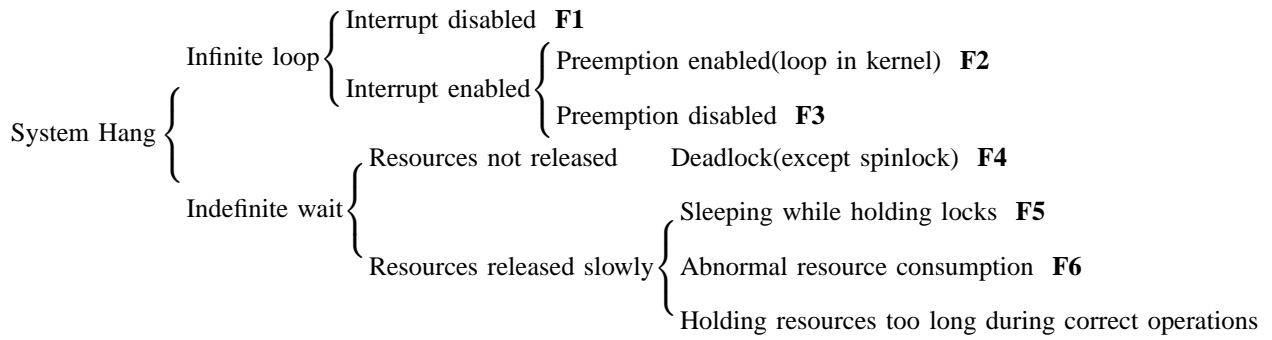


Figure 1. Categories of system hang causes (**F:Fault** in abbreviation)

core computer, the stall of only one core can cause the freeze of the whole system for certain reasons, e.g., the synchronization mechanism between different cores. Indeed, this phenomenon does occur frequently in our experiments.

2.2.2. Indefinite Wait

Awaiting resources (e.g., signals, semaphores, I/O, interrupts or memory spaces) indefinitely can be explained as waiting for the resources requested either infinitely or for a long time (depends on the patience of users). The deadlock described in F4 does not include the circumstance triggered by spinlocks even if double spinlocks (it belongs to F1) is also a kind of deadlock. If tasks or a piece of kernel codes, which have several interactions with other tasks, are trapped by deadlock, system hang may occur due to the sudden loss of the key internal services. In general, sudden disappearance of resources (e.g., peripheral devices, pipe) also belongs to F4. OS provides no mechanism to ensure that a task holding spinlock would not fall into a sleep state. As a result, F5 may cause system hang because tasks that wait for the spinlocks to be released have to run on CPU in a busy waiting way, thus providing no chance to schedule other tasks. F6 is usually relevant to anomalous memory consumption, since there are not enough memory space immediately provided to the new forked tasks or the ones swapped in again. The classical malicious program “fork bomb” (fork infinitely) also belongs to F6. Holding resources for a long time during correct operations, e.g., copying many files simultaneously to peripheral devices, may cause temporal system hang. However, this situation is not considered as a cause of system hang, since it is a correct operation and varies with different system configurations. It should be noticed that although F5 and F6 may release resources after a while (e.g., the task holding spinlock is waked up and executes an unlock operation), F5 and F6 are considered as the causes of system hang because they occur due to inappropriate operations.

3. Empirical Detection Metrics

The difficulty in handling system hang lies in how to detect it, since OS offers no mechanisms to make itself informed when it enters a hang state. Most studies (as described in Section 1) detect system hang

through additional assistances (e.g., hardware modules or kernel modification), this section investigates whether exploiting the services provided by the OS itself can help detect system hang. In Section 3.1, we first introduce a hypothesis about empirical metrics used for system hang detection. According to this hypothesis, the research questions about detection metrics are proposed in Section 3.2. In Section 3.3, we conduct experiments to determine which metrics should be selected to detect system hang. Finally, we discuss how to use the selected performance metrics to detect system hang.

3.1. Hypothesis of Detection Metrics

We choose system performance metrics (e.g., context switches per second and number of runnable tasks) as the targets of detection because they are usually provided by most OSes and implicate the overall performance information when the system slows down. Our detection metrics are hypothesized as follows:

Hypothesis: Combined with a theoretical analysis, partial system performance metrics can be regarded as a sufficient basis to determine whether system hang occurs.

3.2. Research Questions

Since system performance metrics are uncontrollable, it is impossible to build a mapping from performance metrics to a hang state. As a result, the other way, i.e., observing the values of performance metrics when the system enters a hang state, can be attempted to help understand which metrics may implicate system hang. It should be noticed that, in this situation, the influenced performance metrics are necessary rather than sufficient to detect system hang.

As a result, whether the selected metrics are also sufficient or not needs to be validated (empirically in Section 5). According to the hypothesis (Section 3.1) and the analysis above, we seek to answer the following research questions:

RQ1 Among hundreds of system performance metrics provided by OS, which ones should be selected?

RQ2 How to determine system hang with the system performance metrics?

Sections 3.3 and 3.4 answer the two research

questions respectively.

3.3. Which Performance Metrics to Select

In this section, we investigate experimentally which metrics to select to detect system hang by observing if a metric changes abnormally under hang scenario. First, we describe our experimental setup. Then, we use an example to show how these experiments work. Finally, the system performance metrics which have potential to detect system hang are selected according to our experimental results.

3.3.1. Experiment Setup

The six types of faults (see Section 2) that cause system hang are considered as the injected faults, which are implemented as errant kernel modules and loaded dynamically under different workloads. Accordingly, the activation rate of injected faults to cause system hang is 100%. We select 68 system performance metrics (e.g., number of tasks currently blocked and percentage of time spent by soft interrupt requests) as the observation targets. To observe the general variations of performance metrics under sufficient workloads, 9 programs (context1, dhry, ftime, hanoi, shell8, pipe, spawn, syscall, and execl) in the benchmark suite (UnixBench 5.1.2) are selected, which could represent at least 95% of kernel usage [26]. Experiments are performed on two computers. One with Intel Core i5 650, 3.20GHz CPU (seen as 4 CPUs by OS) and 4GB RAM, and the other one with Intel Pentium 4, 3.20GHz CPU (seen as 2 CPUs by OS) and 512MB RAM. We consider a Linux kernel (version 2.6.32) as our experimental operating system. To guarantee the generality of the experimental results, each type of injected faults is loaded and executed under each selected UnixBench workload 10 times in each computer. Consequently, the total number of experiments conducted is $6 \times 9 \times 10 \times 2 = 1080$.

3.3.2. An Example

We choose F5 and inject it in the *pipe* workload of UnixBench running on the computer with Intel Core i5 650, 3.20GHz CPU and 4GB RAM.

Although experienced programmers avoid using semaphores after a spinlock to make an unlock operation executed quickly, they may ignore whether the called functions after a spinlock have operations on semaphores or sleep. As a result, tasks which wait for the spinlock to be released (the task holding the spinlock falls asleep due to the *down()* operation on semaphore or explicitly sleep operation, F5) have to run on CPU in a busy waiting way, leaving no chance for other tasks to run. We inject the sleeping kernel module with a spinlock A at the 23rd second, and inject the kernel modules which acquire A at the 39th, 51st and 59th seconds consecutively. As shown in Figure 2-(a), 2-(b) and 2-(c), metric *sys* (percentage of time spent by system call and exception) reaches and holds 100%, and the value of metric *usr* (percentage of

time spent by application) is still zero after injecting the respective kernel module. Finally, CPU0 cannot execute the user program any more when the system enters a hang state (see Figure 2-(d)). In addition, after the 59th second, the number of context switches per second (*cs*) (as shown in Figure 2-(e)) is small since the other three CPUs are occupied by the injected kernel codes. Although some metrics vary obviously after the injection of the faults, e.g., the number of runnable tasks under the *pipe* workload (Figure 2-(f)), they may not be selected as detection metrics, since the value of influenced metrics may be normal in other workloads (e.g., the number of runnable tasks for the *shell8* workload as shown in Figure 2-(f)).

After injecting F5 into the *pipe* workload 10 times and finishing the experiments of F5 in other 8 workloads of UnixBench, the general detection metrics selected for F5 are *usr*, *sys* per CPU, and *cs*.

3.3.3. Experimental Conclusion

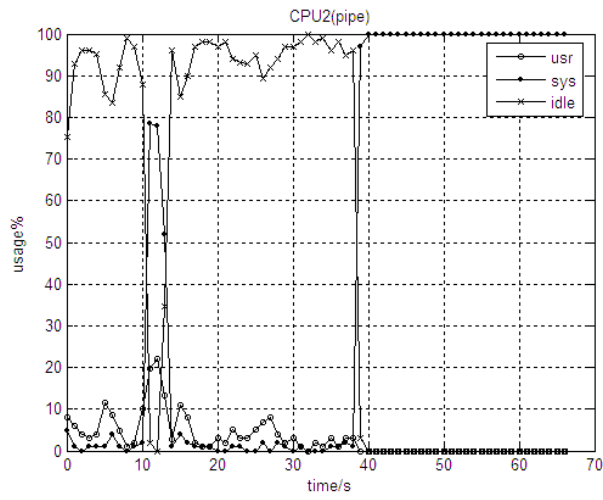
Similar to the methodology adopted by the above example, other experiments are implemented, and the experimental results are given in Table 1. Metric *iowait* represents the percentage of time spent by I/O wait. *run* means the number of tasks in the running state and *blk* records the number of tasks currently blocked. Metric *pswpout* means the number of pages swapped out per second and *memfree* records the unused space of memory. *util* means the percentage of CPU time during which I/O requests were issued to the device. The 9 system performance metrics in Table 1 are considered as the metrics to detect system hang. F1, F2 and F3 have the same detection metrics since they all consume CPU inappropriately. F4 makes the tasks sleep to wait for the services provided by the tasks which are trapped in deadlock, thus it has no influence on the CPU metrics. Because F5 makes the tasks run on CPUs in a way of busy waiting, its metrics are similar to the ones related to infinite loops. As for F6, since it has relevance to consumption of large resources, its detection metrics should be related to memory and I/O.

Table 1. Performance metrics used to detect system hang

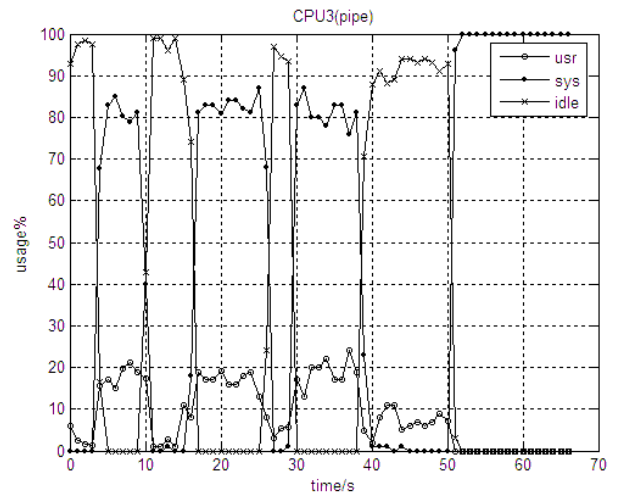
Fault	Metrics		CPU			Process		Memory		disk I/O
	<i>sys</i>	<i>usr</i>	<i>iowait</i>	<i>run</i>	<i>blk</i>	<i>cs</i>	<i>pswpout</i>	<i>memfree</i>	<i>util</i>	
F1	✓	✓				✓				
F2	✓	✓								
F3	✓	✓				✓				
F4			✓		✓					
F5	✓	✓				✓				
F6			✓	✓	✓		✓	✓	✓	✓

3.4. How to Determine System Hang

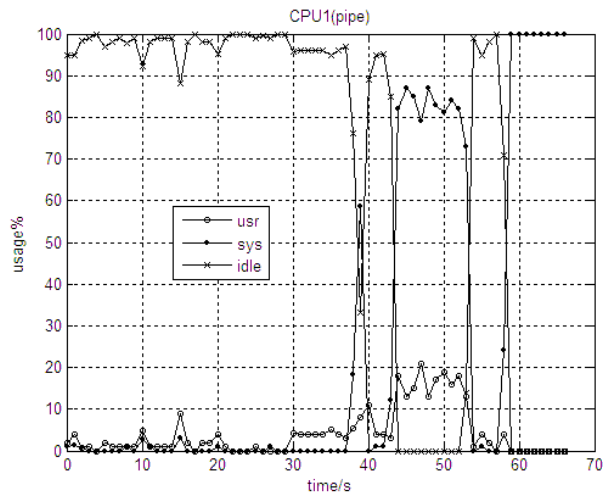
The values of several monitored metrics of system under the normal execution are quite different from those of a hang system. During normal execution, each value of a monitored metric has an acceptable range. The system is considered healthy when each monitored metric is among its acceptable range. By



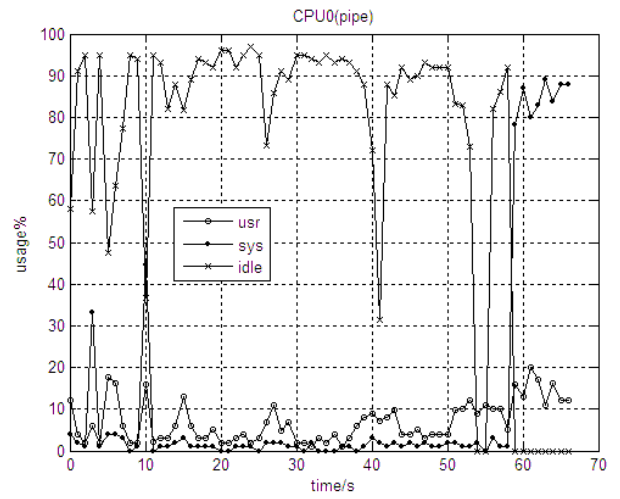
(a) CPU2



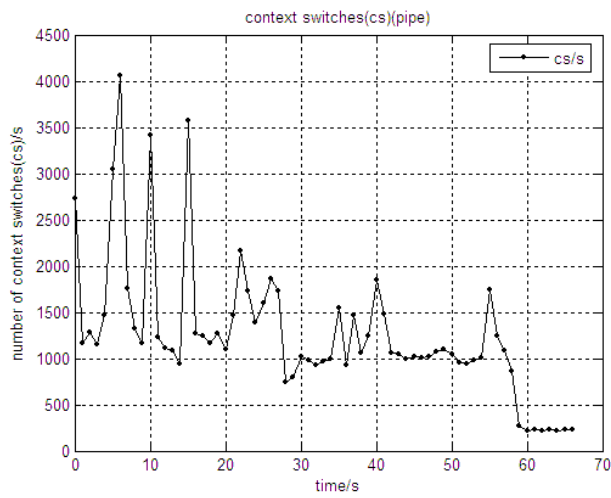
(b) CPU3



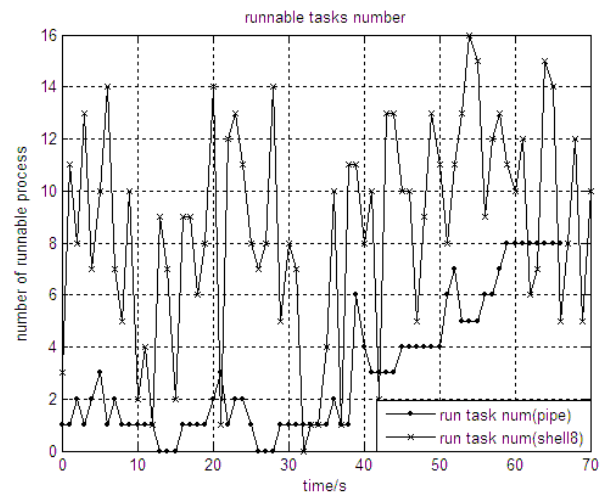
(c) CPU1



(d) CPU0



(e) Number of context switches per second



(f) Number of current runnable tasks

Figure 2. Performance metrics records with F5 in *pipe* workload of UnixBench

comparing the values of metrics collected during the normal and hang states, subject to a further theoretical analysis, we can empirically set an acceptable range for each monitored metric. When the values of one or more metrics are out of their acceptable ranges, we can consider that the system enters a hang state.

For example, after observing the statistics of experimental results, we find that in some hang scenarios *sys* is more than 95% for a long time (e.g., exceeds one second) and *usr* is lower than 4%, which may be caused by a task executing an infinite loop in the kernel mode. However in the normal state, *sys* can hardly reach 90% and last for more than one second, because the time of system calls and exception handling spending on CPU is usually very short. In addition, since the system calls are invoked by user applications, *usr* should not be lower than a certain percentage.

Moreover, the influence of a specific hardware module or operating system should also be considered to improve the portability of the detection strategy of system hang. Thus, an appropriate platform-independent range for each monitored metric is preferred, e.g., some metrics can be evaluated in the form of percentage. In this case, the ranges of some metrics can be initialized according to different hardware and operating system configurations which can be captured when the system starts to run on a specific platform.

4. SHFH: Self-Healing Framework for System Hang

In this section, we introduce SHFH (a self-healing framework to handle system hang), which adopts the methodology of utilizing the 9 empirical system performance metrics (described in Section 3) to detect system hang. To automate the whole process of handling system hang, we introduce the idea of self-healing for designing SHFH. A traditional self-healing architecture includes detection, diagnosis and recovery components [17]. Its diagnosis part is usually implemented into multiple diagnosis engines to capture different failures, which is independent of the detection part. SHFH makes system hang as a failure target and only monitors the performance metrics that may implicate system hang. Its diagnosis mechanism is integrated into the detection component for helping diagnosis-based recovery. This revision remarkably decreases the performance overhead induced by the self-healing framework and simplifies its structure.

4.1. Overview of SHFH

As shown in Figure 3, SHFH contains three core parts: light detector, heavy detector and recovery component.

In SHFH, the light detector only monitors six system performance metrics (see Section 4.2.1) periodically. When it finds that the values of metrics are

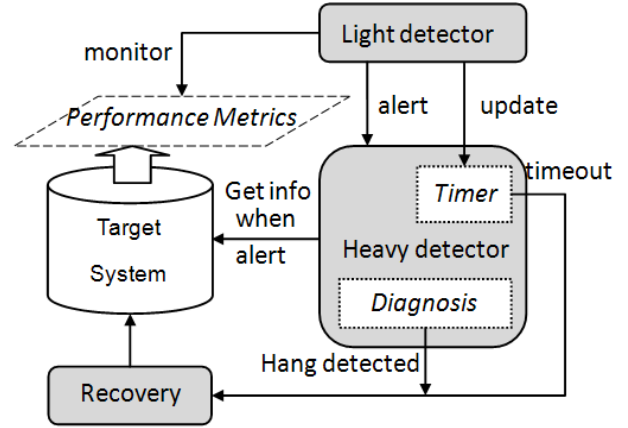


Figure 3. An overview of SHFH

abnormal (perhaps caused by system hang), it triggers an alert to wake up the heavy detector. The heavy detector gets further information by some expensive operations (e.g., poll processes). Then the gathered information is analyzed by the diagnosis part of the heavy detector. If system hang is asserted to occur, the related recovery operations (depending on different diagnosis results), e.g., suspending the current task on a particular CPU or restarting the system, would be executed; otherwise, the alert triggered by the light detector will be ignored by the heavy detector. One unique feature of SHFH is that its “light-heavy” detection strategy is designed to make intelligent tradeoffs between the performance overhead and the false positive rate induced by system hang detection. Because the light detector is lightweight (a user application), expensive operations for collecting further data to detect a hang (to decrease false positives) are incurred (by the heavy detector) only when the light detector triggers an alert.

The light detector is a real time user process, and in some scenarios, it may have no opportunity to be executed due to certain faults (e.g., F1, F2, F3 and F5 described in Section 2). To overcome this problem, a watchdog timer mechanism is introduced in SHFH. The light detector periodically updates the value of the timer in the heavy detector. If the timer is not updated for consecutive periods of time, the services provided by the light detector are regarded as unavailable. Then the recovery operation is called since even the real time application cannot run (there must be something wrong with the system).

4.2. Implementation of SHFH

We have implemented SHFH in the Linux operating system (kernel 2.6.32). The light detector of SHFH is implemented as a real time process, and both the heavy detector and recovery component are implemented as loadable kernel modules. The whole SHFH can be dynamically loaded and removed by simple shell command. In this section, the detailed detection and recovery strategies for system hang, and

the implementations of the light detector, the heavy detector and the recovery component are described.

4.2.1. Light Detector

The light detector can be considered as the eyes of the SHFH, and it is used to collect six system performance metrics, *sys*, *iowait* and *usr* on each CPU, *run*, *cs* and *pswpout* (as described in Section 3.3). According to our formal study (see Section 3.4), system hang can be revealed by the performance metrics. We define some conditions under which an alert should be triggered and at the same time an error code is generated according to different anomalous metrics. The error codes can help the heavy detector perform a further check. The mapping between trigger conditions and an error code is given in Table 2.

Table 2. Mapping model of light detector

Trigger condition	Error code
<i>sys</i> exceeds its upper bound for consecutive monitor interval and the <i>usr</i> does not reach its lower bound	CPU_ERROR
<i>iowait</i> higher than its upper bound	CPU_ERROR
<i>run</i> surpasses its upper bound	PROC_ERROR
<i>cs</i> is lower than its lower bound	PROC_ERROR
<i>pswpout</i> exceeds its upper bound for consecutive monitor interval	MEM_ERROR

The light detector consists of two core functions:

- To obtain the performance metrics of the system, we use the *sar* command to collect data periodically from the */proc* file system which is provided by Linux. By establishing a pipe between the light detector and *sar*, system performance metrics are obtained dynamically.
- Once the light detector finds that some metrics indicate an anomalous condition, it will trigger an alert by sending a message which includes the error code and metrics that are necessary for the heavy detector to perform a further check. Otherwise, it sends an empty message periodically to update the timer of the heavy detector to indicate that the light detector is still working. Sockets are used as the communication medium between the light and heavy detectors.

The light detector acts as a filter of most metrics measured in normal states, which can guarantee that the heavy-cost operations are only executed when the system is in an abnormal state.

4.2.2. Heavy Detector

Unlike the light detector which generates an alert when the system is possibly in a hang state to increase the coverage of hang detection, the heavy detector, which acts as the brain of SHFH, should be able to confirm whether the system is in a hang state to decrease the false positive rate, and then choose a proper recovery action according to different fault causes which can be achieved by a diagnose progress.

The heavy detector can be triggered under one of the two conditions: by receiving an alert message

from the light detector or a timeout signal from the timer that is periodically updated by the light detector. Once triggered, the heavy detector first takes a diagnosis action to check the error code and some extra performance metrics sent by the light detector to confirm whether the system is in a hang state or not. This is necessary because the monitored metrics may also seem anomalous to the light detector under some normal conditions (e.g., the system has a heavy load). Although some performance metrics used to verify system hang are the same as those the light detector, some other metrics are added or the bounds of the metrics are set differently when confirming system hang. The mapping from an error code to metrics for verifying system hang is given in Table 3. Because a recovery strategy is chosen based on the type of faults, the mapping rules from an error code (with extra performance metrics which are sent by the light detector) to the possible faults and recovery actions are also given in Table 3.

Let us consider an example to see how the mapping rules work. As shown in Table 3, when the error code from the light detector is MEM_ERROR, the heavy detector first checks the values of *blk* and *util*, and when both exceed their upper bounds, it polls all tasks to find the task consuming memory abnormally. If this further diagnosis condition is satisfied, F6 is considered as the cause of system hang, and then according to the mapping rules in Table 3, the operation which kills the task that consumes memory abnormally is selected. Otherwise, the heavy detector checks the next mapping rule of MEM_ERROR. If no rules of MEM_ERROR match, the heavy detector will ignore the alert from the light detector.

4.2.3. Recovery

The recovery component of SHFH tries to help OS recover from a hang state and provide continuous services, or restart in some severe cases. Based on different diagnosis results generated by the heavy detector according to the mapping rules, different recovery operations are taken (shown in Table 3). The recovery component offers three types of recovery actions: kill or stop the suspicious process/thread; send an NMI (Non-Maskable Interrupt) to a particular CPU to wake up the stalled CPU; panic the system and then restart. The recovery component may have to restart the OS when the hang scenario is caused by some processes which are in the UNINTERRUPTIBLE state.

5. Evaluation

In order to evaluate SHFH and the effectiveness of the hypothesis described in Section 3.1, we have conducted our fault injection experiments.

5.1. Experiment Setup

The experiments are performed on a computer with Intel Core i5 650, 3.20GHz CPU (seen as 4 CPUs

Table 3. Mapping model of Heavy Detector

Error code	Further diagnosis ²	Diagnosis result	Recovery actions
CPU_ERROR	<i>sys,usr</i>	F1,F2, F3,F5	Send NMI to stalled CPU and kill the running task on the CPU
MEM_ERROR	<i>util, blk</i> and the memory used by each process	F6	Kill the task consumes memory abnormally ³
MEM_ERROR	<i>iowait, pswpout</i> and <i>memfree</i>	F4,F6	Panic and restart
PROC_ERROR	continuous run time of each process	F1,F3	Stop the task runs continuously for a long time ³
PROC_ERROR	<i>run</i> and <i>blk</i>	F6	Panic and restart

by the OS), 4GB of RAM, and a SATA 500GB hard disk with 7200RPM under the Linux kernel 2.6.32. The effectiveness of SHFH is evaluated from three aspects: coverage of fault detection, false positive and performance overhead. The experimental thresholds of detection metrics are presented in Table 4. The rules regarding how to use the thresholds of light and heavy detectors are determined according to their mapping models described in Table 2 and 3 respectively, with the portability of the initial values considered.

Table 4. Experimental thresholds of light and heavy detectors

Light detector		Heavy detector	
Metric	Threshold	Metric	Threshold
<i>sys</i>	50%	<i>sys</i>	95%
<i>usr</i>	4%	<i>usr</i>	1%
<i>iowait</i>	50%	<i>iowait</i>	90%
<i>run</i>	3 * CPU_NR	<i>run</i>	10 * CPU_NR
<i>cs</i>	350	<i>blk</i>	32
<i>pswpout</i>	3000	<i>memfree</i>	128MB
-	-	<i>util</i>	90%

In Table 4, metrics *run* and *cs* have relevance to the number of CPU cores (CPU_NR) seen by OS, and the value of *cs* on each CPU core is about 90. In this experiment, we initialize *cs* as 350 since our experimental computer has 4 CPU cores. The threshold of *memfree* depends on the minimum space of memory reserved by OS. The other initial values are concluded from a theoretical analysis and experiments run with sufficient workloads and different hardware configurations (Sections 3.3 and 3.4). The monitor period of the light detector is set up with one second, which is the shortest monitor period of *sar* with both efficiency and effects being accounted for.

5.1.1. Workload

We have selected Unixbench (Version 5.1.2) and Linux Test Project (LTP Version 2012-01-04) as realistic workloads to evaluate the effectiveness of SHFH. A standard set in Unixbench (same as that in Section 3.3.1) is chosen to generate a normal workload. LTP is used to produce a heavier workload to verify whether SHFH can still work well (considering the false positive rate) under a pressure environment.

5.1.2. Fault Injection

According to the study on the causes of system hang (Section 2), we edit script programs to automatically and randomly inject six types of faults (implemented

as kernel modules) which can lead to system hang into the OS with Unixbench (the 9 test suits as shown in Section 3.3.1 respectively) and LTP running as system overheads. Each type of faults is injected 75 times on Unixbench and LTP respectively to verify whether SHFH can really detect the faults and recover from a hang state. Moreover, some extra experiments with no faults injected are conducted to evaluate the performance overhead and false positive rate. A false positive is considered to occur when SHFH confirms a fault and takes recovery action during normal execution flow of Unixbench and LTP.

5.2. Experimental results

The results of fault injection experiments are shown in Table 5. The number of alert and timeout (generated by the light detector) is larger, while the number of hangs detected (confirmed by the heavy detector) is less than the number of faults injected (75) in some experiments. This shows that in certain environments, the light detector is influenced by the heavy workloads; however, the heavy detector can determine correctly if the system enters a hang state.

Table 5. Results of fault injection experiments

Fault (F1-6)	Workload	Alert/Timeout (times)	Hang detected (times)	Recovery/Panic (times)
F1	LTP	46/29	72	71/1
	Unixbench	32/44	74	74/0
F2	LTP	75/0	72	72/0
	Unixbench	75/0	74	74/0
F3	LTP	30/45	74	72/2
	Unixbench	49/28	75	75/0
F4	LTP	355/0	65	0/65
	Unixbench	280/0	61	0/61
F5	LTP	32/43	68	68/0
	Unixbench	19/55	74	73/1
F6	LTP	104/1	74	71/3
	Unixbench	105/0	75	75/0

As shown in Table 5, in the experiments in which F4 is injected, panic and restart is used to recover in all the cases. That is because when the heavy detector detects the system hang caused by F4, the system has already entered a state that CPU runs normally but other resources (e.g., memory) are not enough, since F4 is usually caused by the waiting tasks which cannot get services provided by the kernel codes or tasks trapped by deadlock. Under this circumstance, system can only be restarted because the deadlock codes or tasks are difficult to locate.

Although the action of panic and then restart (after detecting system hang) is better than powering off the

²Metrics needed to further check to confirm system hang.

³By polling all processes to find the anomalous one.

system, we do not regard it as a successful recovery action. We consider a recovery as successful if it can preserve the operating system with the ability to continue running and providing services after system hang is detected. However, in the worst case, SHFH cannot work due to the serious system hang scenarios (e.g., all CPUs stalled simultaneously). As a result, there are still 4.66% system hang scenarios on average that cannot be recovered and even restarted by SHFH. According to the coverage, false positive, recovery and restart ratio provided by SHFH (shown in Table 6), the effectiveness of the hypothesis proposed in Section 3.1 is empirically validated.

Table 6. Coverage, false positive, recovery and restart ratio provided by SHFH

Work load	Detection coverage	False positive	Recovery ratio	Panic and restart
LTP	94.45%	1.16%	78.67%	15.78%
Unixbench	96.22%	0%	82.44%	13.78%
average	95.34%	0.58%	80.56%	14.78%

The performance overhead is evaluated by the index of system performance which can be captured from Unixbench. By comparing the index result of running a standard benchmark with and without SHFH, we find that SHFH suffers from a performance overhead of about 0.6%. Recall that our experiments are conducted on a multi-core computer. When SHFH is applied on a single-core computer, the detection coverage and recovery ratio may decrease because the recovery operations can not be taken when some types of faults like F1 occur.

6. Related Work

We discuss the related work about the causes of and detection and recovery methods for system hang.

The OS kernel falling into an infinite loop is seen as the reason for system hang [1], [3], [5]; however, that reason may not be appropriate when considering the preemption mechanisms used. Incorrect usage of synchronization primitives (in particular those related to spinlocks in Linux) is regarded as the main causes of system hang [2]. In addition, studies reported in [4], [9] also take into account indefinite wait (for an event that will never occur). However, its effectiveness depends on the way it is waiting for an event (e.g., sleeping or busy waiting).

Several methods have been proposed to detect system hang. The improved watchdog timer [5] needs to be periodically reset under the normal situation; otherwise the timer would expire and an NMI will be triggered. However, this method cannot detect an infinite loop when the process (responsible for resetting the timer) does not get stuck. SHD (System Hang Detector) [1] counts the number of instruction executed between two consecutive context switches. When OS does not schedule processes, the counter value will increase and exceed the theoretical maximum value. This approach is only effective against

an infinite loop with both interrupt and preemption disabled. Monitoring I/O throughput [2] is an effective way to detect some system hang problems, however it fails if a hang occurs within OS code not related to I/O. The work of [4] monitors signals and waiting/holding time for critical sections, task scheduling timeouts, and so on. A total of eight variables for a single process and monitors need to be deployed through dynamic probing with the help of *KProbes* to place breakpoints into the kernel. If applying this strategy into monitoring every process, it may get a sound proof of system hang with low false positives; however, the performance overhead is not optimistic.

Generally, when system hang is detected, restarting the system is regarded as the default recovery action. Study [5] keeps the OS running through killing the current running process. However, when the suspicious process is not the current one, e.g., a process which is sleeping with a spinlock or a large block of memory, the other processes needing the spinlock or the memory space consume the CPU and memory resources and eventually cause system hang. In this case, killing the current process cannot handle system hang. Our recovery strategy varies with the diagnosis results of detection, e.g., killing the sleeping processes (located by the light-heavy detection of SHFH) that hold a large piece of memory which wait for a signal that would never happen, rather than just killing the current process or restarting the system.

7. Conclusion

In this paper, we give a new characterization of system hang according to the two existing views about it, and analyze the causes of system hang in detail from two aspects: indefinite wait for system resources (resources not released or released slowly) and infinite loop under interrupt and preemption constraints. Accordingly, six types of faults that may cause system hang are described. To avoid additional cost incurred by extra assistance (e.g., new hardware modules, kernel modification or breakpoint insertions), we present a hypothesis which only uses a small subset of the set of system performance metrics to detect system hang. Based on this hypothesis, we propose a self-healing framework named SHFH, which can be deployed dynamically, to handle system hang. SHFH can automatically detect system hang and help system recover from it. Evaluation results show that SHFH introduces 0.6% performance overhead and can detect system hang with a false positive rate of 0.58% and a coverage rate of 95.34%, indicating the effectiveness of the “light-heavy” detection strategy adopted in SHFH. Given a recovery rate of 80.56% (making the OS continue running and providing services), its diagnosis-based recovery strategy provides a better recovery granularity than the naive approach that resorts to restarting the system. Finally, our experimental results also validate the effectiveness of our hypothesis that

a small number of performance metrics (9 in SHFH) seem to be sufficient for system hang detection.

Acknowledgement

We thank Roberto Natella and Antonio Bovenzi from Università degli Studi di Napoli Federico II, Haoxiang Lin from Microsoft Research Asia and Zhongkui Sun from Northwestern Polytechnical University for the discussions about causes of system hang with us. This work is supported by Aeronautical Science Foundation of China 20100753022, National Natural Science Foundation of China 61103003 and an Australian Research Council Grant DP0987236.

References

- [1] L. Wang, Z. Kalbarczyk W. Gu and R. Iyer. "Reliability MicroKernel: Providing Application-Aware Reliability in the OS," *IEEE Transactions on Reliability*, 2007, vol.56, pp. 597-614.
- [2] D. Cotroneo, R. Natella, S. Russo. "Assessment and improvement of hang detection in the Linux operating system," In *SRDS*, New York, USA, 2009, pp. 288-294.
- [3] L. Wang, Z. Kalbarczyk and R. Iyer. "Formalizing System Behavior for Evaluating a System Hang Detector," In *IEEE Symp. on Reliable Distributed Systems*, Naples, ITA, 2008, pp. 269-278.
- [4] A. Bovenzi, M. Cinque, D. Cotroneo, R. Natella and G. Carrozza. "OS-Level Hang Detection in Complex Software Systems," *Int. J. Critical Computer-Based Systems*, 2011, vol.2, pp. 352-377.
- [5] F. M. David, J. C. Carlyle and R. H. Campbell. "Exploring Recovery from Operating System Lockups," In *USENIX Annual Technical Conference*, Santa Clara, CA, 2007, pp. 1-6.
- [6] X. Song, H. Chen and B. Zang. "Why software hangs and what can be done with it," In *International Conference on Dependable Systems and Networks*, Chicago, USA, 2010, pp. 311-316.
- [7] N. Nakka, G. P. Saggese, Z. Kalbarczyk and R. K. Iyer. "An Architectural Framework for Detecting Process Hangs/Crashes," In *EDCC*, Budapest, HUN, 2005, pp. 103-121.
- [8] S. Basu, J. Dunagan and G. Smith. "Why did my PC suddenly slow down?" In *Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*, Cambridge, USA, 2007.
- [9] G. Carrozza, M. Cinque, D. Cotroneo and R. Natella. "Operating System Support to Detect Application Hangs," In *VECoS*, Leeds, UK, 2008.
- [10] T. Jarbou, J. Arlat, Y. Crouzet, K. Kanoun and T. Marteau. "Analysis of the Effects of Real and Injected Software Faults: Linux as a Case Study," In *PRDC*, Tsukuba, Japan, 2002, pp. 51-58.
- [11] D. Chen, G. Jacques-Silva, Z. Kalbarczyk, R. K. Iyer and B. Mealey. "Error Behavior Comparison of Multiple Computing Systems: A Case Study Using Linux on Pentium, Solaris on SPARC, and AIX on POWER," In *PRDC*, Taipei, TW, 2008, pp. 339-346.
- [12] X. Wang et al., "Hang analysis: fighting responsiveness bugs," In *EuroSys*, Glasgow, UK, 2008.
- [13] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3.
- [14] R. Love. *Linux Kernel Development*, 3rd. Addison-Wesley Professional, 2010.
- [15] W. Mauerer. *Professional Linux Kernel Architecture*. Wiley Publishing Inc., 2008.
- [16] F. David and R. Campbell. "Building a Self-Healing Operating System," In *DASC*, Columbia, USA, 2007, pp. 3-10.
- [17] H. Psaiar and S. Dustdar. "A survey on self-healing systems : approaches and systems," *Cloud Computing*, 2010, vol.91, pp. 43-73.
- [18] A. Avizienis, J. Laprie, B. Randell, C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing," In *IEEE Transactions on Dependable and Secure Computing*, Los Alamitos, USA, 2004, pp. 11-33.
- [19] I. Lee and R. Iyer. "Faults, Symptoms, and Software Fault Tolerance in Tandem GUARDIAN90 Operating System," In *FTCS*, Toulouse, France, 1993, pp. 20-29.
- [20] W. Gu, Z. Kalbarczyk, and R. K. Iyer. "Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors," In *DSN*, Washington, D.C., USA, 2004, pp. 887-896.
- [21] Sourceforge.net. Linux Test Project (LTP). URL: <http://ltp.sourceforge.net/>
- [22] Google Project Hosting. unixbench-5.1.2.tar.gz. URL: <http://code.google.com/p/byte-unixbench/>
- [23] D. Bovet and M. Cesati. *Understanding the Linux Kernel*, 3rd. O'Reilly & Associates, Inc., 2005, pp. 228-252.
- [24] E. Ciliendo, T. Kunimasa, B. Braswell, *Linux Performance and Tuning Guidelines*, IBM Redpaper, July 2007.
- [25] M. Sullivan and R. Chillarege. "Software Defects and Their Impact on System Availability-A Study of Field Failures in Operating Systems," In *International Symposium on Fault-Tolerant Computing*, Nuremberg, Germany, 1991, pp. 2-9.
- [26] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. "Characterization of linux kernel behavior under errors," In *DSN*, San Francisco, CA, USA, 2003, pp. 459-468.
- [27] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. "An empirical study of operating system errors," In *ACM Symp. Operating Sys. Principles*, New York, NY, USA, 2001, pp. 73 -88.
- [28] N. Palix, G. Thomas, S. Saha, C. Calvs, J. Lawall, and G. Muller. "Faults in Linux: Ten years later," In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, 2011, pp. 305-318.