2009 28th IEEE International Symposium on Reliable Distributed Systems

Assessment and Improvement of Hang Detection in the Linux Operating System

Domenico Cotroneo*, Roberto Natella*†, and Stefano Russo*†
*Dipartimento di Informatica e Sistemistica, Università degli Studi di Napoli Federico II, Via Claudio 21, Naples, Italy
†Laboratorio CINI-ITEM "Carlo Savy", Complesso Universitario Monte Sant'Angelo, Ed.1, Via Cinthia, Naples, Italy
{cotroneo, roberto.natella, stefano.russo}@unina.it

***Abstract*—We propose a fault injection framework to assess hang detection facilities within the Linux Operating System (OS). The novelty of the framework consists in the adoption of a more representative faultload than existing ones, and in the effectiveness in terms of number of hang failures produced; representativeness is supported by a field data study on the Linux OS. Using the proposed fault injection framework, along with realistic workloads, we find that the Linux OS is unable to detect hangs in several cases. We experience a relative coverage of 75%. To improve detection facilities, we propose a simple yet effective hang detector, which periodically tests OS liveness, as perceived by applications, by means of I/O system calls; it is shown that this approach can improve relative coverage up to 94%. The hang detector can be deployed on any Linux system, with an acceptable overhead.**

I. Introduction

Operating Systems (OSs) act as the core of a wide range of computer systems, enabling the development of complex and distributed services. Commodity OSs are being used even in critical scenarios, since they allow a reduction of development time and costs. However, they suffer the well-known dependability pitfalls that characterize Off-the-Shelf (OTS) software modules [1]. A failure in the OS may compromise the correct execution and application performance, as well as the mission of the overall system [2]. Hence, characterizing the dependability of OSs is a major concern.

It has been demonstrated that *software faults* (i.e., bugs) are a major cause of system failures [3]–[5]. As they can manifest transiently, depending on environmental conditions (e.g., hardware and software configuration, workload, and timing events), they often elude testing efforts, thus resulting in failures on the field; this is especially true for OSs, which are very complex and often made up of millions of lines of code, often written by third-party developers (e.g., device drivers). Several research studies have been conducted on fault tolerance issues within commodity OSs, such as replication, checkpointing, driver restarting, and microreboots [6]–[10]. All these mechanisms rely on the OS capability of decting the occurrence of a failure. Failure detection capabilities are a fundamental requirement for enabling fault tolerance strategies, and to achieve self-management in complex and distributed systems [11]. However, there are failures that may occur without being explicitly notified by the system, for which a recovery action cannot be started. As shown by several studies on field failures [4], [12]–[14], detection mechanisms within OSs can fail, and they may lead to missing, misleading, or false detections. This is the case of *hang failures*, which are caused by a significant part of software faults in OS code [15]; they cause the OS to get partially or completely stalled, and to not provide any response to user-space applications. It is difficult to detect hang failures, because they hamper the execution of the OS, and, unlike other types of failures (e.g., illegal memory access), they are not notified by the hardware.

Focus of this paper is on the assessment and the improvement of hang detection mechanisms in commodity OSs. The problem of hang detection has been faced in past research studies (see §II). However, they assume knowledge about workload or availability of special hardware, thus hampering their adoption in commodity OSs. Our driving idea is to propose a fault injection framework to assess hang detection facilities, and to guide their improvement. To the best of our knowledge, there are not fault injection techniques tailored for the assessment of hang detection mechanisms. Fault injection is a valuable approach for experimental evaluation of fault tolerance mechanisms, as it provides the means for studying of complex interactions between faults, errors, failures and fault tolerance mechanisms [16], and it can be used to complement and increase the confidence of other validation approaches [17]. However, existing fault injection techniques and tools, based on bit-flips (e.g., NFTAPE) and generic software faults (e.g., G-SWFIT), are able to induce a hang only in a small minority of cases [18], [19], and no previous work discusses fault representativeness with respect to hang scenarios. In this paper, we propose a fault injection framework for experimental evaluation of OS hang detection mechanisms. The contribution of this paper is twofold:

1) The design of a representative and effective fault injection framework, which is made up of the following phases:
   - *Collection and analysis of software fault reports*. Our study on the Linux OS confirms that concurrency issues are a major cause of hang failures, as observed from previous field data studies [5], [15];
   - *Definition of a set of faults based on the collected software fault reports*;
   - *Identification of fault locations by system profiling*.

   It is worth noting that although several model checking approaches [20] have been proposed, the fault injection framework is complementary to them, since it is difficult to comprehensively model software systems.
2) It proposes a hang detection mechanism that overcomes limitations of hang detectors in the Linux OS. The proposed mechanism tests OS liveness, as perceived

This work has been partially supported by the Regione Campania in the context of the projects L.R n. 5/2002, "Reti di sensori senza filo per l'identificazione ed il tracciamento di target mobili" and Misura 3.17 POR 200/2006, "REMOAM Reti di sensori per il monitoraggio dei rischi ambientali."

1060-9857/09 $26.00 © 2009 IEEE
DOI 10.1109/SRDS.2009.26

by applications, by means of periodic system call invocations, in order to stimulate most important OS subsystems (e.g., memory management, file system, device drivers); failure detection is performed by monitoring of kernel I/O throughput.

An experimental campaign is conducted on Linux OS, using the proposed framework and workloads representative of long-running applications in real-world systems. Experiments show that, in 75% of cases, the kernel is able to detect a hang (see §IV). The proposed detection mechanism has been able to detect 94% of hangs, thus improving detection relative coverage of 19%, with respect to the considered faultload and workloads (see §VI). The detector exhibits low overhead and no false positives, and it does not require in-depth knowledge of system internals.

The remainder of the paper is organized as follows: §II discusses the state of the art in the field of OS hang detection; §III describes the proposed fault injection framework for OS hang detection assessment; §IV describes the experimental analysis we conduct on Linux; §V proposes a mechanism for improving hang detection, which is analyzed in §VI; §VII ends up with conclusions and future work.

II. Background and Related Research

A. Related Work on Hang Detection

Debugging techniques. They use static and dynamic source code analysis to identify hang root causes. In [21], the disk I/O subsystem is modeled analytically; the model is then compared to execution traces, to identify workload conditions under which performance is suspiciously low, and to fix anomalies (e.g., by improving disk I/O scheduling heuristics). In [22], the focus is on *soft hang bugs* (i.e., bugs causing system unresponsiveness); runtime traces are exploited to search for potential hang points within source code, to avoid unnecessary end-user waits. In [23], developers' knowledge about the system is exploited to formulate assertions, and to check the source code for violations. Assertions enforced on the Linux kernel concern memory management errors, temporal ordering of operations, and deadlocks. Debugging techniques are useful to avoid the occurrence of those hangs whose root cause can be easily pinpointed into the source code. However, they are not able to detect bugs that are triggered under complex conditions and that escape code analysis, thus requiring fault tolerance mechanisms.

Hardware monitoring techniques. They use special hardware such as watchdog timers. The timer is periodically reset by the OS in failure free conditions; otherwise, a Non-Maskerable Interrupt (NMI) is triggered to signal that the timer has expired [24]. This mechanism can fail since the kernel is preemptible, hence timers can be reset even during a hang. In [19], [25], hardware instruction counters are exploited to detect whether the processes or the OS are stalled; the number of instructions executed between context switches, or executed in kernel mode, is compared to a threshold to perform detection. In [26], hardware instrumentation is used to remotely inspect the main memory of a node; software counters are used to profile the execution of OS code that should be executed frequently (e.g., context switching, interrupt handling): if the OS is stalled, counters are not updated. Hardware monitoring techniques are characterized by minimal overhead and cannot be affected by software faults. However, hardware support may be not available, and they are not able to detect failures in which the OS is stalled but the monitored events still occur. More flexible software-implemented techniques are needed to cope with these failures. In this paper we consider hang detection mechanisms that do not rely on hardware instrumentation.

Workload modeling techniques. They provide a synthetic model of system performance, in order to detect anomalies due to both applications and OS faults [27]–[29]. OS level metrics are considered (e.g., CPU utilization, paging, I/O operations, and free memory), which are modeled by means of time series analysis and machine learning (e.g., PCA, decision trees). They are particularly effective when the operational profile is known *a priori*. Regression techniques have been proposed in [30] to reduce false detections due to changes in the operational profile; nevertheless, workload modeling approaches are not immune to false positives. Moreover, workload modeling cannot distinguish between OS faults and application faults, therefore it is difficult to correctly apply OS fault tolerance mechanisms.

B. Technical Background

Kernel data collection tools. Several monitoring facilities are provided by the Linux kernel, which have been exploited in this work. In particular, we use *KProbes*[1], which inserts breakpoints in arbitrary binary code locations in charge of triggering user-defined handler functions. Handlers can be used to collect information about internal kernel variables; subsequently, kernel execution is restored. *Kdump*[2] is a tool for failure data collection based on the execution of a secondary kernel, namely *capture kernel*, which is preliminarily loaded into a reserved memory region. When the primary kernel fails, the *capture kernel* is executed; then, it can collect failure data by reading the main memory state.

Built-in hang detection mechanisms. Several hang detection mechanisms are available in the Linux OS, which can be enabled by recompiling the kernel. In particular, the following facilities can be used for hang detection:

- Soft lockup detection, i.e., the kernel detects whether a "canary" task is not scheduled within a timeout;
- Hard lockup detection, i.e., if any CPU in the system does not handles local timer interrupt for longer than a timeout;
- Sleep-inside-spinlock checking, i.e., assertions that verify whether there are spinlocks that have been acquired before calling a "sleeping" function (i.e., a function during which the current thread may block and be preempted by the scheduler);
- Checks on lock API usage, that is: missing lock initialization, release of an already freed lock, release of a lock by a thread or CPU different from the lock holder, lock data structure corruption.

III. Fault Injection Framework

In this section, we design a fault injection framework focused on hang failures. First, we identify requirements of the framework. We then describe a field data study, on which we base a fault library representative of hang-related faults.

[1]*KProbes is included in the main-line Linux kernel since v.2.6.10*

[2]http://lse.sourceforge.net/kdump/

A. Requirements

Past studies on fault injection evidenced that hang failures are difficult to reproduce in a representative and effective way. For instance, in [19], errors at OS and application layers are injected by means of bit-flips; this approach was able to exhibit a small number of hang failures (for Linux and Windows OSs, and Apache Web Server, hangs were observed in the 9.0%, 1.7%, and 0.4% of cases, respectively). Moreover, there are no guarantees that bit-flips actually represent real faults responsible for hang failures. Similar observations can be made for the G-SWFIT technique [18]; in that work, only 18% of experiments cause a hang failure; moreover, the field data study, which is used to define generic fault operators, did not take into account the type of failure that faults being studied may cause.

In this work, *representativeness* and *effectiveness* are considered fundamental requirements, in order to focus on hang failures and to overcome limitations of past works. This means that the injected faults have to be representative of real software faults that surface as hang failures.

B. Field data study

We adopt a field data based approach in order to inject representative software faults. We collected data from discussion groups available on Internet. We focused on discussions dealing with kernel hangs experienced by Linux developers; in particular, 20 discussions were detailed enough to diagnose the fault and related triggering conditions. In many cases, patches have been proposed to fix the bug, which helped us to pinpoint the fault at source code level. The analysis shows that kernel hangs are mainly due to wrong usage of synchronization primitives, in particular those related to *spinlocks* (i.e., locks in which a CPU actively waits). In particular, the main causes of hang failures are:

- Threads attempting to acquire a lock that is already held. For example, when page swap code is called to release physical memory, a deadlock may occur if there is not enough memory for buffer allocation[3];
- Interrupt handling code, e.g., interrupts are masked when the kernel is waiting for a device-driven event[4,5];
- Wrong handling of a set of locks, e.g., locks related to page tables and memory descriptors should be acquired in a given order before memory management operations[6,7];
- Wrong assumptions about locks held when executing a piece of code, e.g., a function should not be called without releasing a specific lock[8].

In order to gain further insights about hang-related faults, we analyze comments within Linux source code. Comments are often used by developers to describe a fix they made, therefore they are useful to spot hang-related bugfixes not described in discussions. Moreover, developers use comments to describe under which assumptions their code works correctly; because the Linux source code is very large and complex, and several developers work simultaneously on it, they need to know how to use the code developed by others, by means of comments and documentation. If a developer disregards these assumptions, a fault will likely occur; therefore, comments provide information on the faults that can affect the kernel. We extract comments from the Linux source by searching for keywords related to hang failures (e.g., "hang", "freeze", "stall", "deadlock"); 2565 source code comments were found and individually analyzed; from them, 147 comments were related to hangs due to software faults; we classify them in the following hang failure causes[9]:

1) A lock is acquired by a thread that already holds it (35 comments). This may happen in the case of recursive functions, and in the case of two functions calling each other, which use the same lock.
2) A set of two or more locks is improperly managed (20 comments). In these cases, comments describe in what order locks should be acquired, in order to avoid deadlocks (namely, A-B-B-A deadlocks).
3) A thread should possess the correct set of locks before calling a function (59 comments). For example, before calling a function that may "sleep", it should release all held spinlocks, which have to be acquired again when the function wakes up. Because a programmer may not know all sleeping functions, comments are used to avoid faults.
4) Wrong handling of interrupt state (39 comments). For example, there are spinlock primitives that have to be used when the CPU state (e.g., interrupt masking on a CPU) has to be preserved before entering a critical section; the state can be restored when releasing a lock. Interrupt masking may cause the stall of a thread on that CPU.

C. Fault injection design

Injection technique. From the analysis of hang failure causes, we formulate a set of faults able to reproduce them. Because there exist several approaches for emulating software faults, we discuss our design choices and existing fault injection approaches with respect to three fundamental aspects [31] (i) type of faults to inject (*what*), (ii) fault location within the OS (*where*), and (iii) time at which to inject faults (*when*).

There are two fundamental approaches that can be adopted. The first approach, namely Software Mutation (SM), emulates software faults by injecting bugs into the source or binary code of the target system. The second approach, namely Software-Implemented Fault Injection (SWIFI), corrupts the internal state of a running program to emulate errors, with no mapping to a software fault (e.g., bit-flip injection [32]). In this work, SM is preferred to SWIFI (*what* to inject), since it provides the most accurate emulation of software faults [16]. Moreover, SM implies that faults should be injected within the OS code (*where* to inject), and that a fault should be present for the whole duration of an experiment (*when* to inject); since software faults are permanent, injection of representative faults can not be made at an arbitrary time [16].

Fault library. We introduce a set of source mutations, in charge of reproducing all hang failure causes previously identified (§III-B). The *fault library* is composed by four mutations (one for each failure cause):

[3]http://lwn.net/Articles/261271
[4]http://thread.gmane.org/gmane.linux.kernel/286491
[5]http://lwn.net/Articles/138165
[6]http://article.gmane.org/gmane.linux.kernel.mm/16003
[7]http://lwn.net/Articles/130160
[8]http://lwn.net/Articles/274292

[9]6 comments were classified under two causes.

F1 Missing release of a spinlock that has been acquired;
F2 Locking of a set of two or more spinlocks in wrong order;
F3 Missing unlock/lock pair on the same spinlock;
F4 Omission of interrupts state restoring.

Faults from the library consist in modifying or removing existing kernel code that uses synchronization primitives; this emulates *missing* and *wrong* constructs, which have been demonstrated to account for the majority of real faults from the field (respectively 64.2% and 33.1% of faults collected in [18]). In practice, it is unlikely that programmers insert extra faulty code into a program. Differently from G-SWFIT, which uses generic fault operators, we focus on code that uses synchronization primitives and can cause hang failures.

Fault locations. The number of potential fault locations in a complex OS can be very large; among them, there are several source locations rarely executed or with low probability of fault activation. We inject faults into source locations that provide a high probability of fault activation, with respect to the considered workloads. In this way, we increase the effectiveness of injection campaigns.

To choose a proper subset of fault locations, we preliminarily profile their invocations, by instrumenting the kernel. In particular, we measure the number of times each spinlock has been acquired, as well as the number of times it has been contended by two or more threads. Then, we inject faults in locations for which a spinlock:

- has been acquired more than N times; this increases the probability that fault locations will be executed.
- has been contended by other threads more than M times; faults will more likely be activated and surface as a hang.

Figure 1 summarizes the steps to be followed to inject hang-related faults in the kernel. Based on the results of our field data study, the fault library can be used to inject fault extensively, wherever there are invocations of spinlock API.

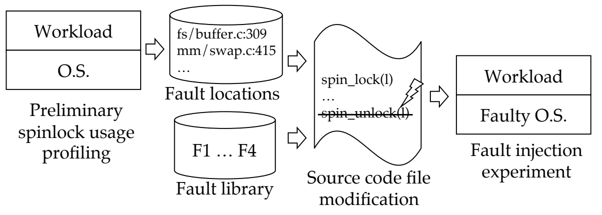

Figure 1. Fault injection phases.

IV. Assessment of Linux Detection Mechanisms

A. Workload

We evaluate hang detection mechanisms with respect to a set of realistic workloads, which are representative of long-running applications. To this aim, we use FileBench[10], an open source framework for measuring filesystems performance. FileBench provides predefined workload profiles; we consider the following workloads:

1) `Varmail` (**W1**), which emulates a Network File System (NFS) mail server. The workload consists of a multi-threaded set of open/read/close, open/append/close and delete in a single directory.
2) `Fileserver` (**W2**), which performs a sequence of create, delete, append, read, write, and attribute change operations on the filesystem; the workload uses a big set of files and nested directories.
3) `Oltp` (**W3**), which is a database emulator. This workload reproduces an I/O model similar to DBMSs. It tests the performance of small random reads and writes, which are the typical access pattern of OLTP databases. It extensively uses OS Inter Process Communication (IPC) facilities to synchronize readers, writers, and DB processes.

Since workloads are multithreaded, concurrency mechanisms are stimulated significantly. This is helpful to reproduce environmental conditions in which hangs occur.

We modify default FileBench profiles by introducing two workload states, to evaluate detection mechanisms with respect to more realistic scenarios. In the first state (CPU-I/O bound), the standard FileBench profile is executed, which emulates periods of busy workload; in the second state (CPU bound), the workload is composed only by CPU bound processes, which emulate periods of I/O inactivity. The workload resides in a given state for a random time; in both states, the residence time is a Normal random variable, with $\mu = 5$ min and $\sigma = \mu/10$. The total duration of each workload amounts to 1 hour.

We performed experiments on a computer equipped with 2 Xeon 2.8GHz CPUs with Hyper-Threading (4 CPUs are seen by the OS), 5GB of RAM, and a 36 GB SCSI hard disk. Kernel version 2.6.25 has been considered, and workloads have been generated by FileBench version 1.64. Fault injection and lock profiling have been made by instrumenting spinlock primitives at source code level; it should be noted that the fault injection framework does not require source code availability, since binary instrumentation techniques can be also used [18].

B. Faultload

Workloads are profiled to identify fault locations (§III-C); we assume $N = 1000$, $M = 100$. Table I summarizes fault locations identified. The last column of the table shows the percentage of faults leading to kernel hangs; the overall percentage of activated faults (52.07%) is greater than the percentage of hang failures reported in past works (see §III-A). We conclude that fault injection campaigns can actually benefit from the proposed framework, which can improve the effectiveness of experiments.

Table I
Fault locations.

Fault type	W1	W2	W3	All	Activated	
F1	37	38	33	108	42	38.89%
F2	4	4	4	12	6	50%
F3	3	4	3	10	8	80%
F4	13	13	13	39	32	82.05%
All faults	57	59	53	169	88	52.07%

Table II shows the distribution of injected faults across OS subsystems; these subsystems account for the most of code executed by the OS, therefore the fault injection campaign will cover failures of the most important subsystems, which are actually used by the workloads.

[10]http://www.solarisinternals.com/wiki/index.php/FileBench

Table II
INJECTED FAULTS DISTRIBUTION.

OS subsystem	F1	F2	F3	F4
Block I/O Layer	3.28%	0%	3.28%	0%
Disk Driver	3.28%	0%	3.28%	0%
VFS Layer	13.11%	8.20%	1.64%	0%
EXT3 Filesystem	19.67%	1.64%	0%	4.92%
Process Management	14.75%	0%	6.56%	1.64%
Memory Management	8.20%	0%	6.56%	0%

C. Results

In order to evaluate hang detection mechanisms, we use well-known metrics [33]: *mistake rate* (Γ) and *detection time* (Λ). Γ is defined as the mean number of false alarms per second, and it is estimated by counting the number of false alarms during faulty-free executions. Λ is defined as the interval between the occurrence of a failure and the first failure notification. Moreover, since failures may not be detected, we evaluate the *coverage* (Ψ), i.e., the probability of detecting a failure given that it has occurred. Ψ is estimated by the ratio between the number of detected failures and the number of failures actually occurred during fault injection.

Results in table III evidence that Linux mechanisms failed to detect hangs to a significant extent. About the 25% of injected faults resulted in hangs that have not been detected by the kernel; this holds with respect to individual workloads and to the whole set of experiments.

Table III
COVERAGE OF LINUX DETECTION MECHANISMS.

Fault type	W1	W2	W3	All
F1	92.86%	100.00%	100.00%	97.62%
F2	100.0%	100.00%	100.00%	100.00%
F3	33.33%	33.33%	50.00%	37.50%
F4	50.00%	36.36%	44.44%	43.75%
All faults	70.97%	77.42%	76.92%	75.00%

The best results have been obtained with respect to classes F1 and F2. They are related to the repeated acquisition of a lock, hence they are eventually detected by means of a check into the spinlock primitive that acquires that lock.

The analysis of fault types shows that Linux detection mechanisms are able to detect fault types F3 and F4 only in a subset of cases. This fact implies that the location in which faults are injected affects Ψ. By means of in-depth analysis, we discovered that faults F3 and F4 are detected when a thread encounters an assertion before hanging. Figure 2 shows an example of assertion; before invoking a sleeping function (line 9), the function checks (line 3) that interrupts are not masked on the current CPU, and that no spinlocks are held, i.e., thread preemption is safe. Therefore, when the kernel makes a wrong use of spinlock primitives, the assertion will notice an error state, producing a warning message.

When assertions in kernel code are inserted, developers need to foresee potential error propagation paths; however, there can be hang failures escaping these assertions. For example, in Figure 3, the `prepare_to_wait()` function is used within the kernel to put the current thread in a queue to wait for a condition to be true. When fault F4 is injected in this function (lines 10-11), interrupts are not re-enabled after a thread executes it. Unless other threads on that

```
 1 void writeback_inodes(struct writeback_control *wbc) {
 2    struct super_block *sb;
 3    might_sleep();        // ASSERTION
 4    spin_lock(&sb_lock);
 5    // ... OMISSIS ...
 6       if (down_read_trylock(&sb->s_umount)) {
 7          if (sb->s_root) {
 8             spin_lock(&inode_lock);
 9             sync_sb_inodes(sb, wbc);      // SLEEPS
10             spin_unlock(&inode_lock);
11          }
12          up_read(&sb->s_umount);
13       }
14    // ... OMISSIS ...
15 }
```

Figure 2. writeback_inodes() (fs/fs-writeback.c:510)

CPU re-enable interrupts, or incur in a `might_sleep()` assertion, they will hang with no error signal.

```
 1 void prepare_to_wait(wait_queue_head_t *q,
 2                      wait_queue_t *wait, int state) {
 3    unsigned long flags;
 4    wait->flags &= ~WQ_FLAG_EXCLUSIVE;
 5    spin_lock_irqsave(&q->lock, flags);
 6    if (list_empty(&wait->task_list))
 7       __add_wait_queue(q, wait);
 8     if (is_sync_wait(wait))
 9       set_current_state(state);
10    ~~spin_unlock_irqrestore(&q->lock, flags);~~
11    spin_unlock(&q->lock);  // FAULTY
12 }
```

Figure 3. prepare_to_wait() (kernel/wait.c:66)

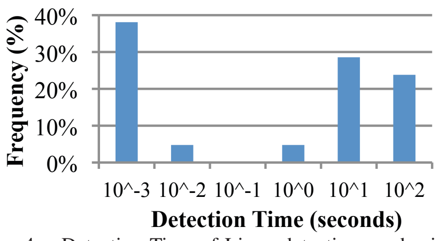

Figure 4. Detection Time of Linux detection mechanisms.

As for Λ, Figure 4 shows the distribution of detection delays for detected hangs. We observe that the most of hangs has been detected within 1 ms once the faulty code has been executed. However, there is a percentage of failures that have been detected in a much longer time (more than 1 s). This is due to the fact that much time can pass before a kernel thread incurs in an assertion. Therefore, under worst-case scenarios, the system can be unavailable for a long period of time.

V. THE PROPOSED HANG DETECTOR

The experimental results (§IV) show that there are hang scenarios in which the Linux OS is unable to detect the failure. In fact, these scenarios occur when an injected fault violates assumptions made by developers, and there are not internal checks able to detect such violations.

It is difficult to foresee all underlying assumptions made by programmers, because of the great complexity of OS source code. Therefore, it is useful to augment internal checks with global detection mechanisms, to detect when the OS is in a hang state regardless of the location in which the hang occurs. The proposed approach is a global detection mechanism, which tests OS liveness by applications' point of view. In fact, if a hang occurs within the OS, it will impact on the availability of one or more OS services provided to applications. Therefore, by testing if an OS service is actually provided, we can detect the occurrence of a hang; the test is made by monitoring OS performance.

In this work, we consider I/O throughput as reference performance metric. This metric is important because, in fact, several critical and long-running applications (e.g., DBMSs, Web Servers) are I/O bound or mixed CPU-I/O bound. Moreover, I/O services are tied to several OS subsystems (e.g., memory management, filesystem, device drivers) accounting for the most of OS code. In this context, I/O throughput measurements can be used to detect several hang scenarios. Instead, this approach can fail if a hang occurs within OS code not related to I/O (e.g., the IPC subsystem); however, if the workload makes extensive use of such code, other performance metrics can be adopted.

I/O throughput is obtained by measuring the amount of data transferred by device drivers (e.g., blocks read from a hard disk). The OS is seen as a two-states system, respectively *idle* and *active*; the system is idle if I/O throughput is null, active otherwise (Figure 5). Formally,

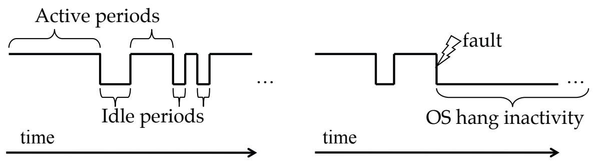

Figure 5. I/O activity during faulty-free and faulty runs.

we can represent I/O throughput as a random process $X(t)$, where $y(t) = 1$ if $X(t) > 0$ (active state), and $y(t) = 0$ otherwise (idle state). A hang failure is detected at time t, when the I/O driver has been in idle state for more than C consecutive time units (time units have been assumed equal to 1 s); this can be expressed as $y(t-j) = 0$, for $j = 0 \ldots C$.

Although I/O bound workloads frequently invoke I/O services, there can be periods of idle workload (e.g., web traffic changes) in which no I/O requests are issued. Moreover, workload failures, such as a stall due to a fault in user-space applications, may cause a device driver to be in idle state even if a kernel hang is not occurred. To discern between OS hangs and workload idleness or failures, we include in the workload a background process (namely, *heartbeat process*) that periodically tests I/O services, with a period H equal to or less than the detection timeout, by means of filesystem calls (e.g., read, write, open, seek) on private files. The heartbeat process is simple and well-tested, in order to guarantee that it is fault-free.

The detector is implemented as a Linux kernel module periodically scheduled by a timer. The detector monitors the interface between drivers and the kernel, since the most of commodity OSs provide well-defined programming interfaces that can be used by third-party extensions, along with profiling tools for monitoring them (such as *KProbes*, *DebugView*[11] for Windows, and *DTrace*[12] for Solaris). Therefore, the approach can be also adopted on other commodity OSs. The proposed detector does not require hardware instrumentation, and does not rely on assumptions about the workload; therefore, it can be easily deployed on systems running the Linux OS.

[11]http://technet.microsoft.com/en-us/sysinternals/bb896647.aspx
[12]http://www.sun.com/bigadmin/content/dtrace/

VI. Assessment of the Proposed Detector

In this section, we evaluate the proposed hang detector, with respect to the same workloads and faultload of §IV. We assume $C = 5$ seconds for the detection module, and $H = 5$ seconds for the heartbeat process. Table IV shows Ψ for the hang detector; for all tested workloads, it is greater than the Ψ of Linux detection mechanisms (table III), with an overall improvement of 19.32%. This improvement is due to the fact that the proposed hang detector does not rely on the source location in which a failure manifests, thus overcoming limitations highlighted by injection of faults F3 and F4.

Table IV
Coverage of the proposed detector.

	W1	W2	W3	ALL
All faults	90.32%	96.77%	96.15%	94.32%

However, there is still a small part of failures not detected by the proposed hang detector. In these hang scenarios, interrupts are disabled on all CPUs, thus hampering execution of the hang detector. To isolate the hang detection module from OS failures, hardware support can be exploited whereas available (e.g., the hang detector can be triggered by means of periodic NMIs instead of timer interrupts).

Moreover, the proposed hang detector provides an upper bound for Λ (unlike existing detection mechanisms, Figure 4), since hangs are eventually detected by the timeout mechanism after C time units. We actually observe a bounded Λ for all the injected faults; therefore, we conclude that Λ solely depends on C.

The timeout C also impacts on Γ. Indeed, the greater its value, the better is Γ (i.e., the lower mistake rate). It is worth noting that C has no impact on Ψ (i.e., if the failure can be detected by the proposed approach, it will be eventually detected for any timeout choice). Therefore, we evaluate Γ with respect to C, for each workload (Figure 6); again, we assume $H = 5$ seconds. For short values of C, we observe a high Γ, due to idle periods occurring in mixed CPU-I/O bound workloads. Γ sharply decreases for $C = H = 5$, and $\Gamma = 0$ for $C > H$; hence, increasing C over $H + 1$ does not improve Γ. A similar behavior has been also observed for other values of H.

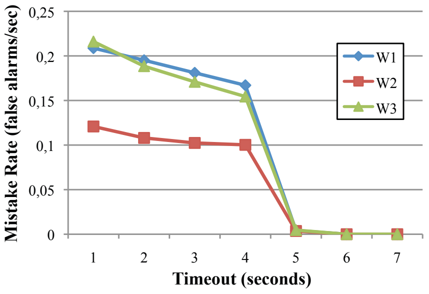

Figure 6. Mistake Rate of the proposed detector.

We also evaluate the overhead of the proposed detector; table V compares workload performance without and with the hang detector, for $H = 1$ s. We analyzed FileBench results, namely number of I/O operations and transferred MBs per second, and mean duration of I/O operations. It is shown that the performance penalty is less than 5%, even for a low value of H. We also made a hypothesis test, namely T-test, to verify if differences between performance are statistically significant; in all cases, there is not a significant difference between mean values, with a confidence of 99% (the p-value in the table is always greater than 0.01). Therefore, the proposed detector can be applied for OS monitoring, and the heartbeat period can be set reasonably low.

Table V
OVERHEAD OF THE PROPOSED DETECTOR.

Workload	ops/s		MBs/s		latency	
	% diff.	p-value	% diff.	p-value	% diff.	p-value
W1	1.83%	0.1137	1.43%	0.0617	1.69%	0.0890
W2	0.09%	0.8913	0.13%	0.8367	4.01%	0.0614
W3	3.27%	0.1371	1.91%	0.4313	4.16%	0.0958

VII. CONCLUSIONS

In this work, we faced the problem of assessing and improving hang detection in commodity OSs. In particular, we proposed a fault injection framework to evaluate hang detection mechanisms systematically. Focus was on Linux OS, which is widely used even in critical application contexts. The injection framework, based on a field data study, is specifically focused on hang failures, and it provides greater representativeness and effectiveness than existing fault injection approaches.

The injection framework highlighted limitations in existing detection mechanisms of Linux OS, which were taken into account to design a hang failure detector; it increased the relative coverage from 75% to 94%, with respect to realistic workloads. It also exhibits low overhead and avoids false positives, thus it can be deployed on any Linux distribution. Moreover, we believe that it can be adopted on other commodity OSs, as it does not require in-depth knowledge of OS internals. Future work encompasses the development of more robust OS hang detection mechanisms, with respect to faults in the OS, based on virtualization techniques.

REFERENCES

[1] E. Weyuker, "Testing component-based software: A cautionary tale," *IEEE software*, vol. 15, no. 5, pp. 54–59, 1998.

[2] R. Moraes et al., "Experimental Risk Assessment and Comparison Using Software Fault Injection," in *Proc. IEEE Intl. Conf. on Dependable Systems and Networks*, 2007.

[3] J. Gray, "A census of Tandem system availability between 1985 and 1990," *IEEE Trans. on Rel.*, vol. 39, no. 4, 1990.

[4] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Networked Windows NT System Field Failure Data Analysis," in *Proc. IEEE Pacific Rim Intl. Symp. on Dependable Computing*, 1999.

[5] M. Sullivan and R. Chillarege, "Software Defects and their Impact on System Availability: A Study of Field Failures in Operating Systems," in *Proc. IEEE Intl. Symp. on Fault-Tolerant Computing*, 1991.

[6] G. Candea et al., "Microreboot - A Technique for Cheap Recovery," in *Proc. Symp. on Operating Systems Design & Implementation*, 2004.

[7] M. Swift et al., "Recovering device drivers," *ACM Trans. on Computer Systems*, vol. 24, no. 4, 2006.

[8] M. Baker and M. Sullivan, "The recovery box: Using fast recovery to provide high availability in the UNIX environment," in *Proc. USENIX Summer Conf.*, 1992.

[9] B. Cully et al., "Remus: High Availability via Asynchronous Virtual Machine Replication," in *Proc. USENIX Symp. on Networked Systems Design and Implementation*, 2008.

[10] F. David and R. Campbell, "Building a Self-Healing Operating System," *Proc. IEEE Intl. Symp. on Dependable, Autonomic and Secure Computing*, 2007.

[11] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, vol. 36, no. 1, 2003.

[12] C. Simache and M. Kaaniche, "Availability Assessment of SunOS/Solaris Unix Systems based on Syslogd and wtmpx log files: A case study," in *Proc. IEEE Pacific Rim Intl. Symp. on Dependable Computing*, 2005.

[13] M. F. Buckley and D. P. Siewiorek, "VAX/VMS event monitoring and analysis," in *Proc. IEEE Intl. Symp. on Fault-Tolerant Computing*, 1995.

[14] A. J. Oliner and J. Stearley, "What Supercomputers Say: A Study of Five System Logs," in *Proc. IEEE Intl. Conf. on Dependable Systems and Networks*, 2007.

[15] A. Chou et al., "An Empirical Study of Operating System Errors," in *Proc. ACM Symp. on Operating Systems Principles*, 2001.

[16] M. Hiller, J. Christmansson, and M. Rimén., "An Experimental Comparison of Fault and Error Injection," in *Proc. IEEE Intl. Symp. on Software Reliability Engineering*, 1998.

[17] J. Arlat et al., "Fault Injection for Dependability Validation: A Methodology and Some Applications," *IEEE Trans. on Soft. Eng.*, vol. 16, no. 2, 1990.

[18] J. Duraes and H. Madeira, "Emulation of Software Faults: A Field Data Study and a Practical Approach," *IEEE Trans. on Softw. Eng.*, vol. 32, no. 11, 2006.

[19] L. Wang et al., "Reliability MicroKernel: Providing Application-Aware Reliability in the OS," *IEEE Trans. on Reliability*, vol. 56, no. 4, 2007.

[20] L. Wang, Z. Kalbarczyk, and R. Iyer, "Formalizing System Behavior for Evaluating a System Hang Detector," in *IEEE Symp. on Reliable Distributed Systems*, 2008.

[21] K. Shen, M. Zhong, and C. Li, "I/O System Performance Debugging Using Model-driven Anomaly Characterization," in *Proc. USENIX Conf. on File and Storage Technologies*, 2005.

[22] X. Wang et al., "Hang analysis: fighting responsiveness bugs," in *Proc. EuroSys Conf.* ACM, 2008.

[23] D. Engler et al., "Checking system rules using system-specific, programmer-written compiler extensions," in *Symp. on Operating Systems Design & Implementation*, 2000.

[24] F. M. David, J. C. Carlyle, and R. H. Campbell, "Exploring Recovery from Operating System Lockups," in *USENIX Annual Technical Conference*, 2007.

[25] N. Nakka et al., "An Architectural Framework for Detecting Process Hangs/Crashes," in *Proc. European Dependable Computing Conference*. LNCS, 2005.

[26] F. Sultan et al., "Recovering Internet Service Sessions from Operating System Failures," *IEEE Internet Computing*, vol. 9, no. 2, 2005.

[27] D. Pelleg et al., "Vigilant: Out-of-Band Detection of Failures in Virtual Machines," *Operating Systems Review*, vol. 42, no. 1, 2008.

[28] W. Linping et al., "A proactive fault-detection mechanism in large-scale cluster systems," *Proc. IEEE Intl. Parallel and Distributed Processing Symp.*, 2006.

[29] Z. Zheng, Y. Li, and Z. Lan, "Anomaly localization in large-scale clusters," *Proc. IEEE Intl. Conf. on Cluster Computing*, 2007.

[30] L. Cherkasova et al., "Anomaly? Application Change? or Workload Change? Towards Automated Detection of Application Performance Anomaly and Change," *Proc. IEEE Intl. Conf. on Dependable Systems and Networks*, 2008.

[31] J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates Software Faults based on Field Data," in *Proc. IEEE Intl. Symp. on Fault-Tolerant Computing*, 1996.

[32] T. Jarboui et al., "Analysis of the Effects of Real and Injected Software Faults," in *Proc. IEEE Pacific Rim Intl. Symp. on Dependable Computing*, 2002.

[33] W. Chen, S. Toueg, and M. K. Aguilera, "On the Quality of Service of Failure Detectors," *IEEE Trans. on Computers*, vol. 51, no. 5, 2002.