Quantitative evaluation of fault propagation in a commercial cloud system

Chao Wang and Zhongchuan Fu

Abstract
As semiconductor technology scales into the nano regime, hardware faults have been threats against computational devices. Cloud systems are incorporating more and more computing density and energy into themselves; thus, fundamental research on topics such as dependability validation is needed, in order to verify the robustness of clouds for sensor networks. However, dependability evaluation studies have often been carried out beyond isolated physical systems, such as processors, sensors, and single boards with or without operating system hosts. These studies have been performed using inaccurate simulations instead of validating complete cloud software stacks (firmware, hypervisor, operating system hosts and workloads) as a whole. In this article, we describe the implementation of a fault injection tool, which validates the dependability of a commercial cloud software stack. Hardware faults induced by high energy density environments can be injected; the fault propagation through the cloud software stack is traced, and quantitatively evaluated. Experimental results show that the integrated fault detection mechanism of the cloud system, such as fatal trap detectors, has left a detection margin of 20% silent data corruption to narrow down. We additionally propose two detection mechanisms, which proved good performance in fault detection of cloud systems.

Keywords
Cloud computing, dependability evaluation, fault injection, cross-layer resilience, intermittent fault

Date received: 4 October 2019; accepted: 6 January 2020

Handling Editor: Mohsin Iftikhar

Introduction
The increasing use of wireless sensors that generate massive data combined with the need to process these data efficiently has given an enormous importance to the cloud computing paradigm. While the basic components of IoT (Internet of Things) systems have limited storage and processing capabilities, they have usually seamless access to the Internet which allows them to profit from the abundant (and usually low-cost) resources available in the cloud.

Reliability of the cloud is a major concern, considering its intensive use and remote communication nature. A cloud system should provide the services it is designed for with robustness; this fact induces additional challenges to its design and development.

Verification, validation, and testing techniques should be used to ensure reliability of these systems.

There are many kinds of reliability threats that cloud platforms need to face. According to the source of threats, we classify them into two categories: external threats and intrinsic threats. External threats include electric failures, system overload, system

Corresponding author:
Zhongchuan Fu, Computer Department, Harbin Institute of Technology, Harbin 150001, China.
Email: fuzhongchuan@hit.edu.cn
misconfiguration, security threats and attacks, and natural disasters (e.g., Hurricane Sandy). Intrinsic threats include software problems (e.g., immaturity of the software stack) and hardware failures. Unlike the external threats, the errors or faults (which may finally lead to a failure, as depicted in the “Background” section) originated from the internal threats generally have an incubation period, before failures manifest themselves. This may bring more uncertainty to the system’s fault detection, and consequently, has been a permanent and serious concern to platform maintainers.

In addition, hardware failures manifest more frequently in the electronic devices as semiconductor technology scaling into the ultra-deep submicron regime. The combinational logics in processors make them more vulnerable, compared to the sequential logic. An experiment toward a DLX superscalar architecture shows that nearly 90% of errors came from combinational logic units. An error report by Rashid et al. from 950,000 personal computers shows that approximately 39% of the reported hardware errors about microprocessors are intermittent. Therefore, we adopt the intermittent fault as our evaluation model with combinational logic units (and register file) as fault injection targets.

Moreover, we believe that the dependability of the software stack of a cloud system remains unclear. The reason is that failures induced by intrinsic threats are hard to trace, especially during the simulation step at the granularity of instructions. This granularity may provide comprehensive information to explore the fault propagation in both hardware and software, in comparison with message-level fault injectors (e.g., ORCHESTRA), only collecting messages such as local time and number of received messages. Accordingly, implementation of an instruction-level fault injector for cloud systems might help researchers to model the propagation and identify the root cause of the intrinsic failures in a cloud, and this is helpful for fault detection and diagnosis methods design.

In order to validate the dependability of a cloud against hardware faults, we propose to perform fault injection simulations and use the cloud computing system as a target. From a design perspective, it would have been a very complex task to develop a cloud simulator from scratch, with the requirement of a full-system simulator, that is to meet the need of supporting interrupt handlers, boot loader, memory management, and other functions to run operating systems (OSs). In addition, we also needed to simulate the cloud firmware and the hypervisor in order to run multiple OS hosts upon; this posed an extra weight in terms of complexity on the system design. As a benefit, only dynamic instructions, that is, instructions that have been currently executing in the CPU pipelines, unlike static fault injection technologies, are injected and traced and ultimately logged during the simulation. This ensures the exclusion of static instructions from the faulty trace and guarantees the analytic accuracy of fault modeling by minimizing the redundancy of each trace. We set up a Fault Behavior Tracing (FBT) system integrated with a full-system simulator, based on which a commercial cloud system runs inside. Using FBT, we simulate the hardware faults originated from inside of the processor and monitor the fault behavior of the cloud software stack.

In this article, the vulnerability of a cloud system against hardware faults is evaluated, and the propagation of faulty instructions through the software stack is quantitatively studied. The failure circumstances of the cloud system are traced and modeled. Based on this study, we have also implemented two fault detection mechanisms to narrow the dependability margin that state-of-art cloud systems have left over.

This article is organized as follows. The “Background” section presents a short background on validation techniques to check reliability of clouds. The “Design and implementations” section presents the design and implementation of our simulation tool, named FBT; the fault injection methodology is also described in this section. The “Experimental results” section presents the validation experiment. Based on the result analysis, fault detection mechanisms are proposed and implemented in FBT, and the detection performance is also evaluated. The “Related work” section discusses related work, and the “Conclusion” section concludes the article.

Background

In this section, we introduce the terminology on system dependability used throughout this work. The fault models used in the reliability validation experiment in the “Experimental results” section are also presented, through a selective survey of the presence and impact of semiconductor devices lifetime in computing systems.

Terminology

The basic definitions of dependability of computing systems can be traced to 1982 in Lee and Morgan and have been discussed by Avizienis et al. and Salfner et al. In this article, we will use the concepts summarized by Kondo et al. where they describe the basic threats to reliability, namely failures, errors, and faults:

- **Failure**: An observable event that occurs when the system deviates from its correct state and therefore it fails to operate as planned.
- **Error**: The part of the total state of the system that may cause a failure. When an error does not cause an external failure, it is a dormant error. Those which provoke a failure are active errors.
Hardware faults may have multiple effects on the state of programs at runtime, for example, the silent data corruption (SDC), mask, and benign fault:

- **SDC**: Silent data corruption, a fault that has not been detected by hardware, OS, or other mechanisms in the system, and ultimately leads to program errors.
- **Mask**: A fault that has been aborted during the propagation process across layers of the cloud system.
- **Benign fault**: A benign fault is a fault that has not been masked but allows a program to continue execution after the occurrence of the fault.

**Hardware fault models**

Premature advances in technology have impelled a resurgence of interest in intermittent faults. The driving forces include shrinking geometries, smaller interconnected dimensions, lower power voltages, and decreased noise margins, all of which have a negative impact on chips’ lifetime and yield. The underlying physical mechanisms make intermittent faults characterized as follows:

- **Burst**: Unlike transient faults due to single-event upset (SEU), intermittent faults come from wear-out, as age and wear take their toll on the electronic device. As going deeper in age until the wear-out phase, the fault begins to occur in bursts, each of which ranges from orders of cycles timescale to milliseconds or even seconds.
- **Irregularity**: Intermittent faults are expected to occur under some particular situations, such as temperature increase, voltage drops, noise perturbations, and so on. This indicates that intermittent faults can hardly be modeled in some distribution, which is named irregularity.
- **Repair**: An intermittent fault usually manifests in the same subunit of the device. Replacement of the fault component would eliminate the intermittent fault.

**Design and implementations**

In this article, we aim to evaluate the vulnerability of a cloud computing system quantitatively. Our scheme is to set up a full-system simulator, based on which we inject hardware faults into dynamic instructions in CPUs, and model the propagation behavior by tracing the instruction sequence until the faults are detected or the workloads are accomplished. In this section, we introduce the implementation of our FBT system. We use the FBT simulator based on the well-known software asset management (SAM) to simulate the considered case studies. The fault injection methodology configured in the experiment is also described.

**Derivation of fault injector’s implementation level**

Since there are multiple levels for fault injectors, and designs at different levels may have different effects on performance and accuracy, we need to make some trade-offs between levels, before a simulator is implemented. Some fault injection tools based on ASICs (application-specific integrated circuit) or FPGAs (field-programmable gate array) can simulate faults with excellent fidelity. However, ASIC-implemented fault injection tools have limited fault injection precision (i.e. irradiation of the ASIC chip with a laser source in Samson et al.). The register-transfer-level (RTL) validation tools also have limited simulation performance that are not suitable for large commercial software workloads. Preliminary work shows that the booting sequence of the Solaris OS may take tens of years to complete, when using an FPGA-implemented tool.

Similarly, software-implemented simulation tools focus on the verification of CPU prototypes and performance evaluation, such as SimpleScalar, which has implemented simulation of super scalar prototypes at the instruction level. Other tools such as Simics, GEMS, and GEM5 focus on architectural performance verification; NS3 meanwhile is a network simulation platform. Some of the simulators mentioned above are full-system simulators, but none of them supports virtualization technologies such as the virtual machines (VMs) and cloud infrastructure. Our FBT is based on SAM, which is a full-system simulator, designed to assess the feasibility of virtualization environment. Aiming at evaluating the reliability of cloud systems, we implemented (1) fault injection modules used to inject faults into target locations and to sustain fault during a certain time; (2) fault models, needed as a dictionary indicating the fault types (e.g. bit-flip, stuck at), sustain duration, and burst length (for intermittent faults); (3) controller, to start/terminate/restart the fault injection campaigns, and recognize infinite loop and blue screen; and (4) trace monitor, responsible for recording each instruction into the trace file and necessary information such as privilege mode, exceptions, core number, and so on, and other components needed by a fault propagation tracer. FBT supports the running of firmware, hypervisor, VMs, GuestOS, and user applications on top of it, as well as configurations for a cloud system that are supported in SAM.
**Architecture of FBT**

FBT is set up to simulate fault propagation from individual units of processor throughout the software stack in the cloud computing system. Currently, the simulator supports fault injections into the address generator, decoder, arithmetic logic unit (ALU), floating-point unit (FPU), and register files in the processor, and monitoring trace from the instruction buffer and state registers (seen in the left side). Three types of hardware faults can be injected, and software is simulated with the granularity of single instructions. The software stack is monitored by FBT modules we developed (in red brown color in Figure 1): monitor interface, trace monitor, trace analyzer, controller, and fault behavior collector.

The overall architecture of FBT is described in Figure 1. FBT consists of two main components: the software operating environment, and the reliability verification component. The former simulates the whole software stack of the cloud computing environment, on the full-system simulator SAM, with a single-instruction granularity. The reliability verification modules treat SAM as a fault injection target. First, through the analysis of each module of SAM, we have performed code-level modification of the processor simulator, including the addition of the fault injection interface and the monitor interface, which implements fault injection for multiple units in the processor. The fault injection interface has the purpose of destructively striking the instructions that stay in each unit. For example, for the multi-bit flip fault of the decoder, the instruction field and the operand field are re-decoded, according to the flip position by the fault injection module, and then the result of the re-decoded instruction is sent to the next instruction in the instruction sequence. At the same time, the trace monitor records the instructions (before and after modification). In this way, the fault injection and trace tracking functions of a simulation step with granularity of a single instruction are setup.

Furthermore, we designed and implemented the trace analyzer module to obtain comparison of traces. For example, when the fault occurs within the same fault location but in different time (instructions or data), the trace analysis module shows the differences in fault propagation. In the processor of FBT, functional units such as ALU, FPU, decoder, AGEN, and register file have been selected as fault injection targets. The specific fault injection locations in each individual injection target are described in Table 1. For the decoder unit, we perform the fault injection by perturbing the opcode or operand field of the target instructions; the fault injection location should be the decode buffers. Different from decoder, the faults cannot be

---

**Table 1.** Target units and corresponding fault locations.

| Units     | Fault locations                      |
|-----------|--------------------------------------|
| Decoder   | Instruction decode buffers           |
| ALU_FPU   | Input latch of ALU and FPU           |
| AGEN      | Output latch of address generation unit |
| Register file | Bits in register file               |

Decoder: decoder unit, which is responsible for decoding instructions and generating control signals. ALU_FPU: integer arithmetic logic and floating-point unit. Note that both the integer and floating-point instructions’ format is parsed in FBT, and thus the evaluation covers the operations in ALU and FPU. AGEN: address generation unit, a key unit for instruction sequence used to generate the memory address of the next executed instruction. FBT: Fault Behavior Tracing.

---

**Figure 1.** The architecture of FBT (Fault Behavior Tracing system).
injected by simply the operand field of the (floating-point) arithmetic logic instructions for the ALU and FPU unit, because SAM simulates the prototype of the UltraSPARC T2 processor, which is a RISC instruction set (SPARC V9) processor. The operand field of (floating-point) arithmetic logic instructions is defined to be a register number. In FBT, the fault injection of ALU and FPU is implemented by parsing the register number from the target instruction, and then modifying the data of the specific register. Through source code analysis, we found that the NPC register (Next Program Counter register) is used to indicate the address of the next instruction to be fired in the SAM simulator. Hence, for the AGEN unit, we modified the instruction address in the NPC register right after the current instruction is written back. The injection of the register file is also a modification operation to the target register, which is similar to ALU and FPU, but the modification does not need to be synchronized with any instructions being executed. Therefore, there is a relatively high probability of the masked faults during fault propagation (see the “Experimental Results” section for details).

Due to the millions of experiment instances that are necessary in order to ensure the confidence interval of the experimental results, we need to inject faults automatically. We implemented the blue screen recognition and the dead loop detection in the controller module, to recognize the system crashes which are caused by illegal memory address access, trap stack overflow, and other serious failures.

**Fault injection methodology**

Fault models for transient faults, intermittent faults, and permanent faults are listed in Table 2. The fault location (i.e. position of the faulty bit) in each target is selected randomly and follows a uniform distribution. Note that the simulation of intermittent faults, as depicted in Figure 2, consists of three parameters: fault activation interval ($T_A$), fault inactivation time interval ($T_I$), and burst length ($L_{burst}$). At RTL level, it has been validated that the separation between fault activations ($T_I$) does not influence failures rates under the assumption of fixed activations ($T_A$).

The timescale of each burst is specified as orders of instructions, due to the implementation level of our fault injection tool. Moreover, in accordance with the characteristic of the intermittent fault mentioned in the “Background” section, intermittent fault injection in FBT is designed as follows: (1) **Burst**. Each burst of fault generates a faulty instruction and the fault parameter burst length ($L_{burst}$) can be specified as two, four, eight, and sixteen instructions respectively in FBT; (2) **Irregularity**. The burst number for intermittent fault in each emulation instance is randomly selected; (3) **Repair**. The fault location remains unchanged throughout one emulation instance in the experiment.

We gathered statistics for a 2-month period fault injection tests of the Mibench workloads, including BasicMath, Dijkstra, FFT, Qsort, and StringSearch. For each instance, one of the three types of fault is chosen and injected into the target fault location. For intermittent faults, an overall of 24,000 runs (300

**Table 2.** Fault models in this work.

| Fault types       | Fault models | Descriptions                                      |
|-------------------|--------------|--------------------------------------------------|
| Transient faults  | bit-flip     | In storage elements, representing SEU (single-event upset) |
| Permanent faults  | stuck@0      | In combinational logic or storage elements       |
|                   | stuck@1      | In combinational logic and interconnection lines |
| Intermittent faults | bit-flip    | In storage elements, SEU occurs intermittently between two logic values 0/1 |

**Figure 2.** Intermittent faults configuration including faults parameters: $T_A$, $T_I$, and $L_{burst}$. $T_A$: fault activation interval; $T_I$: fault inactivation time interval; $L_{burst}$: burst length, # of activation intervals.
injections \(\times 4\) units \(\times 5\) benchmarks \(\times 4\) \(L_{\text{burst}}\) is conducted; for transient faults and permanent faults, we conducted 12,000 and 6,000 instances respectively, since there are two types of faults with permanent stuck@0 and permanent stuck@1 faults, compared to transient faults with only one. The simulator is in 1c1t (1 core 1 thread) configuration, as the multi-core configuration is left for future work.

**Experimental results**

In this section, we evaluate the dependability of the software stack against hardware faults for cloud computing systems. We use the FBT simulator based on SAM to simulate the considered case studies.

**Dependability evaluation of the software stack for a cloud computing environment**

Hardware faults may have no effect on the state of the program at runtime, and ultimately yield correct result (called a masked fault); or they may lead to a state corruption. In the latter case, if the state corruption is not detected by a fault detection mechanism, and if it fails to be masked during the propagation process either, it may result in an SDC. Hardware faults can be masked by the following reasons: logical masking, latch window masking, and electrical masking. However, the effectiveness of these masking mechanisms tends to decline due to the shrinking feature size of the semiconductor device. From the fault injection campaign, we found that the integrated hardware fault detection (except for user-customized redundancy mechanisms) is still dominated by fatal traps in a commercial cloud system.

As shown in Figure 3, for all fault injection instances, the fatal trap accounted for 42.87%, the masked faults accounted for 38.06%, and the SDC accounted for 19.06%. In the case of removing the IRF file, the proportion of fatal traps further increased, which accounted for 56.93% (fatal trap), 20.50% (masked), and 22.56% (SDC). The IRF file is sequential logic, and the symptom distribution is obviously different from the other three units. The masked faults reach up to 90.75%, and the SDC is 8.55%, while the fatal trap is less than 1%. The reasons for this difference are as follows: (1) The faulty data in the register have been rewritten before it is referenced, and (2) the faulty data are referenced by a dynamic dead instruction, which shows the positive impact of the uncertainty of data correlation on the fault shielding ability.

Furthermore, we distinguished the privilege level of the instruction triggered by the fatal trap and divide it into application fatal trap (in non-privileged mode) and OS fatal trap (in privileged mode). The ratio of application fatal traps is 88.13% (50.21%/56.93%), compared to OS fatal traps, which accounts for less than 12%.

Note that application fatal traps may trigger OS fatal traps during execution, and these traps are not counted in. Therefore, this ratio does not represent frequencies of all the fatal traps but may represent the probability that a fault propagates into this privileged mode and is detected in the same privileged mode. For example, the AGEN unit is dominated by 0x9 and 0x10 traps in case of a fault. The 0x10 trap (illegal_instruction trap) mostly occurs when the Program Counter (PC) register is changed by the fault, causing miss in Cache (0x9) or memory. When the processor enters HP mode (hyper-privileged mode) to execute the service routine of 0x9 trap, the fault continues to disrupt the value of the PC register, causes the instruction address to fall into the non-code area in memory, and further triggers the 0x10 trap. In such case, we record this fault detected by 0x9 trap in non-privileged mode.

The 20.50% proportion of the masked faults indicates fault shielding capabilities throughout the cloud system in FBT. Once a fault has been injected into an instruction, it will cause an architectural state corruption directly, unlike the lower level fault injection tools, which may lose the corruption probability during the fault propagation from circuit level to architecture level. It is important to remind that we did not implement FBT in lower level also to ensure the cloud software stack to be simulated in affordable time.

The possible reasons of the masked faults are the following: (1) the fault is injected in the target field of the branch instruction, but the branch jump does not occur; (2) the sign bit of an arithmetic operation determines the branch instruction to jump, but the error result does not affect the sign bit; (3) the fault changes the opcode, but the final result remains unchanged, such as “add %o3, 1, %o3” becomes “or %o3, 1, %o3.” If the value
of the o3 register is even, the result of the instruction operation is not affected; and (4) the NOP instruction is injected into the Sethi instruction. Since the destination register of this instruction is g0, the value of the register is not affected, and so on.

SDC indicates the fault which will lead to erroneous results. For the three combinational units, the average SDC has reached 22.56%, while the register file is lower but has still reached 8.5%, which shows the margins the software stack has left over for other detectors to narrow. We have implemented two other detectors in FBT, which have been incorporated into FBT and performed well in reducing the proportion of SDC (see the “Proposed detection mechanisms” section).

**Dependability analysis of fault time variations**

Figure 4 shows the variation of symptoms along with the duration of the faults (X-axis). For transient faults, the masked faults have the highest proportion, but as the fault duration increases, the share of mask symptoms becomes lower, and it is minimized at the permanent stuck@1, indicating that the fault detection mechanism achieves good performance in case of physically irreversible hardware faults. Note that the SDC rate is slightly lower at permanent stuck@0 compared to the permanent stuck@1; since the default value of most architecture-level states is 0, this fault type may cause architectural state corruption with lower probability.

**Proposed detection mechanisms**

To reduce the SDC, which reaches up to 20% in the cloud environment, we designed two fault detectors and integrated them into the FBT system. The evaluation proved good detection performance of the proposed mechanisms.

**High detector.** In addition to system calls and only a few interrupt handlers that introduce many continuous OS instructions, only tens or hundreds of instructions are executed continuously in most OS modes. However, in case of a fault, such as a fault in the PC register (or a memory load instruction), an incorrect instruction (or data) address is produced. Since the wrong address is accessed, the processor must enter in the HP mode to handle the cache miss (0x9 trap), introducing a large number of OS instructions.

Since there are no such detectors, we integrated the high detectors in FBT. The detector uses a performance counter which is common in the microprocessor to accumulate the number of continuous instructions executed in the P mode (or in the HP mode). When the preset threshold is exceeded, the detector will produce an alarm. Accordingly, when the high symptom is detected in P mode, there is a high OS fault detected; and if detected in HP mode, we call it a high hyper fault. The high detector shows a limited performance overhead.30

The detection results are shown in Figure 5. After adding the high detector, the SDC rate decreased from 20.49% to 4.45%, and shows that the detection effect of the high detector was obviously positive. Among them, the high OS rate contributed with 15.6% and the high hyper rate contributed with 0.43%; however, we believe that when SAM is extended to more VMs, the high hyper detector will show better detection performance as more interactions in hypervisor will be raised.

**Hang detector.** In addition to high detector, we also implemented another detector called hang detector. The hang detector records the number of branch instructions executed in the privileged mode and determines whether it exceeds the profile threshold (normal maximum \( \times 10 \)) to detect an infinite loop, taking a small performance overhead (e.g. 1.6% in distributed OS31). The effect of hang is not obvious, and only few instances have been detected in the experiment. However, taking account the harmful results induced by the hang symptom, like dead loop or SDC after long-term delay, the hang detector is needed.
Latency of the detectors

Detection latency is a crucial factor for detection mechanisms. Long time delays may have negative impact for the reliability strategy to recovery or even make the state irreversible. In this work, the latency is described as the number of instructions from architectural state corruption to fault detection. The injected faults that have been masked in their propagation to the registers are discarded because we believe it is unnecessary to consider the simulation from combinational logic to registers at the architecture level, which may be mainly considered in lower levels, such as FPGA or ASIC design. As shown in Figure 6, faults that have been detected by fatal trap detectors are detected in less than 40 instructions in average. Over 58.9% of intermittent faults are detected within 10 instructions and 97.8% within 20 instructions. About 73.7% of symptoms are detected within 1K instructions, and 97.7% within 100K instructions. This can be handled with hardware checkpointing schemes,\textsuperscript{32} while the higher latency cases can be handled with software checkpointing schemes. The state transition diagram of the fault injection methodology and fault detection mechanisms in the FBT system is shown in Figure 7.

Moreover, an interesting phenomenon has been found: SDCs caused by the transient fault have the highest proportion, exceeding the probability of 1/3 (34.6%). It shows that for the cloud system, the evaluation proved good detecting performance of the proposed mechanisms. Even if there are more software levels to improve the probability of the masked faults and fatal traps detection compared to environment only with OS hosts, the system still needs to be enhanced in order to deal with transient fault sources such as crosstalk.

Related work

Large-scale field studies on failure characteristics of individual components,\textsuperscript{33,34} such as DRAMs and hard disks, have been investigated in real-life cloud systems. The high-level characteristics of hardware failures along with trends and patterns are observed based on statistics of repair events. Google\textsuperscript{34} found that DRAM error behavior differs from commonly held assumptions; that is, memory errors are dominated by hardware errors, rather than software errors, and environment (i.e. temperature) only has a very small effect on error rates.

Wang et al.\textsuperscript{35} propose a fault detection method to make use of symptoms in the RTL level, such as core status (DeadLock) and core-memory events (Invalid Packet) in order to detect intermittent faults. Rashid et al.\textsuperscript{36} make a preliminary study of intermittent fault propagation at the program level. Arasteh\textsuperscript{37} also focuses on the programming level and investigates the inherent effects of different programming structures on the program resiliency. Mueller-Gritschneider et al.\textsuperscript{38} set up a processor simulator supporting cross-layer...
evaluation, but unlike FBT, it is used for soft-error resilience and embedded systems.

Fault injection experiments have been conducted to measure the reliability of OSs. Some of them are able to emulate upsets in FPGA fabric and microprocessor cache lines and special registers. In Le and Tamir43 and Pham et al.,44 fault injection tools implemented in cloud environments are proposed. The authors use them to evaluate and characterize dependability of OSs, taking advantage of virtualization (Virtual Machine Monitor) for fault injection into non-virtualized systems (GuestOS). As a result, the fault injection modules can only be implemented in the Virtual Machine Monitor, and only misbehaviors of the GuestOS can be studied.

I. Feinbube et al. propose a fault injection method45 and a stress-oriented dependability testing model46 to ensure cloud reliability assessment, which relies on exchangeable executables when particular fault types are injected, and may result in limited coverage of the space of possible faults.

Yamato et al.,47 proposed a light-weight restoration method on OpenStack, promising effectiveness in restoring all virtual resources within a short time when physical servers or VM processes are down. The method is implemented in software and can be used after a fault is detected. Thus, this method can be complementary to our method. Some other approaches for assessing cloud system reliability and security have been also explored in Yuan et al.,48 Herscheid et al.,49 Zimba et al.,50 and Wang et al.51

However, as stated in Cheng et al.,52 “resilience to errors in the underlying hardware is a key design objective for a large class of computing systems, from embedded systems all the way to the cloud.” To the best of our knowledge, our work is one of the few that sets up dependability models and the simulator of cloud systems, in order to understand failure behaviors originated in different levels of the CPU or bit-flips in registers. We believe that assessment based on FBT as presented in this article can provide more realistic dependability models, considering the dynamic and cross-layer simulation environments of real-world cloud systems.

Conclusion

This article makes an attempt to investigate the possibility of exploiting a fatal trap as a candidate symptom to protect a cloud system against hardware faults, taking into account the dependability of the software layers in the runtime dynamic execution environment. In order to do this, a fault behavior tracing tool, FBT, has been set up on top of a full-system simulator. Based on FBT, we were able to inject faults at the architecture level (such as bit-flips in registers), and simulate fault propagation through a realistic software stack, including firmware, hypervisor, Solaris OS, and workloads.

The fault behavior has been quantitatively analyzed with respect to fault types, durations, and other factors. Assessment results show that the SDC has reached more than 15% in total in case of a cloud system under hardware faults attack. Then, a high- and hang-based fault detector has been proposed and incorporated into the cloud environment, which has reduced the SDC down to 4.45%. Considering the near zero hardware overhead, the performance improvement of the proposed method is obvious. Going one step further, error propagation patterns of fault models can be extracted from logs, and can be used as diagnosis essay and applied to make failure predictions.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work is partly supported by the Natural Science Foundation of Beijing (Grant No. 4174091), Research Funds for Education Committee of Beijing (Grant No. KM201711232013), and Key Research Project of Beijing Natural Science Foundation (Grant No. Z16002).

ORCID iD

Chao Wang https://orcid.org/0000-0003-3056-0255

References

1. Wolf W. Cyber-physical systems. Computer 2009; 42(3): 88–89.
2. Chaari R, Ellouze F, Koubia A, et al. Cyber-physical systems clouds: a survey. Comput Netw 2016; 108: 260–278.
3. Yu W and Fan L. Vehicular Ad Hoc networks. In: S Misra, I Woungang and S Chandra Misra (eds) Guide to wireless Ad Hoc networks (Computer Communications and Networks). London: Springer, 2009, pp.503–526.
4. Ghobadi A, Karimi R, Heidari F, et al. Cloud computing, reliability and security issue. In: 16th international conference on advanced communication technology (ICACT), Pyeongchang, South Korea, 16–19 February 2014, pp.504–511. New York: IEEE.
5. Maurice G, Diaz F, Coti C, et al. Downtime statistics of current cloud solutions, 2014, http://iwmcr.org/wp-content/uploads/2014/03/downtime-statistics-current-l.3.pdf
6. Kwok N and Kin PC. Process integration, devices and structures. In: P Cogez, M Graef, B Huizing, et al. (eds) The international technology roadmap for semiconductors. Albany, NY: SEMATECH, 2012, pp.25–26.
7. Saggese GP, Vetteth A, Kalbarczyk Z, et al. Microprocessor sensitivity to failures: control vs. execution and combinational vs. sequential logic. In: Proceedings of international conference on dependable systems and networks (DSN 2005), Yokohama, Japan, 28 June–1 July 2005. New York: IEEE.

8. Saggese GP, Wang NJ, Kalbarczyk Z, et al. An experimental study of soft errors in microprocessors. IEEE Micro 2005; 25(6): 30–39.

9. Rashid L, Pattabiraman K and Gopalakrishnan S. Intermittent hardware errors recovery: modeling and evaluation. In: 2012 ninth international conference on quantitative evaluation of systems (QEST), London, 17–20 September 2012. New York: IEEE.

10. Dawson S, Jahanian F and Mitton T. ORCHESTRA: a fault injection environment for distributed systems. In: International symposium on fault-tolerant computing, Sendai, Japan, 25–27 June 1996. New York: IEEE.

11. Lu Q, Farahani M, Wei J, et al. LLFI: an intermediate measure interrupt-blocking times. In: 2015 IEEE international conference on software quality, reliability and security (QRS), Vancouver, BC, Canada, 3–5 August 2015. New York: IEEE.

12. Sangchoolie B, Pattabiraman K and Karlsson J. One bit is (not) enough: an empirical study of the impact of single and multiple bit-flip errors. In: 2017 47th annual IEEE/IFIP international conference on dependable systems and networks (DSN), Denver, CO, 26–29 June 2017, pp.97–108. New York: IEEE.

13. Lee PA and Morgan D. Fundamental concepts of fault tolerant computing: progress report. In: Proceedings of the annual international symposium on fault tolerant computing, Santa Monica, CA, 22–24 June 1982. New York: IEEE.

14. Avizienis A, Laprie JC, Randell B, et al. Basic concepts and taxonomy of dependable and secure computing. IEEE T Depend Secure 2004; 1(1): 11–33.

15. Salfner F, Lenk M and Malek M. A survey of online failure prediction methods. ACM Comput Surv 2010; 42(3): 1–42.

16. Kondo D, Javadi B, Isoup A, et al. The failure trace archive: enabling comparative analysis of failures in diverse distributed systems. In: 2010 10th IEEE/ACM international conference on cluster, cloud and grid computing, Melbourne, VIC, Australia, 17–20 May 2010. New York: IEEE.

17. Christopher KW, Huynh KD, Roarabaugh VM, et al. Method and system for utilizing benign fault occurrence to measure interrupt-blocking times. US5301312A Patent, 1994.

18. McPherson JW. Reliability challenges for 45nm and beyond. In: 2006 43rd ACM/IEEE design automation conference, San Francisco, CA, 24–28 July 2006. New York: IEEE.

19. Oracle VM Server for SPARC 3.3 Release Notes, Part No. E62362, 2015, https://docs.oracle.com/cd/E64670_01/.

20. Samson JRJ, Moreno W and Falquez F. Validating fault tolerant designs using laser fault injection (LFI). In: Workshop on defect & fault-tolerance in VLSI systems, Paris, 20–22 October 1997. New York: IEEE.

21. Ahmed N and Tehranipoor M. Improving transition delay fault coverage using hybrid scan-based technique. In: 20th IEEE international symposium on defect and fault tolerance in VLSI systems (DFT 2005), Monterey, CA, 3–5 October 2005. New York: IEEE.

22. Burger D, Austin T and Bennett S. Evaluating future microprocessors: the simple scalar tool set. Technical report, University of Wisconsin–Madison, Madison, WI, 1996.

23. Magnusson PS, Christensson M, Eskilson J, et al. Simics: a full system simulation platform. Computer 2002; 35(2): 50–58.

24. Martin MM. Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset. ACM Comp Ar 2005; 33: 92–99.

25. Binkert N, Beckmann B, Black G, et al. The GEM5 simulator. ACM Comp Ar 2011; 39(2): 1–7.

26. Bartolozzi L, Pecorella T and Fantacci R. ns-3 RPL module: IPv6 routing protocol for low power and lossy networks. In: International ICST conference on simulation tools & techniques, Desenzano del garda, 19–23 March 2012. Brussels: ICST.

27. Gil D, Saiz L, Gracia J, et al. Injecting intermittent faults for the dependability validation of commercial microcontrollers. In: IEEE international high level design validation & test workshop, Incline Village, NV, 19–21 November 2006. New York: IEEE.

28. Guthaus MR, Ringenberg JS, Ernst D, et al. MiBench: a free, commercially representative embedded benchmark suite. In: Proceedings of the IEEE workshop on workload characterization, Austin, TX, 2 December 2001. New York: IEEE.

29. Shivakumar P, Kistler M, Keckler S, et al. Modeling the effect of technology trends on the soft error rate of combinational logic. In: Proceedings of the international conference on dependable systems and networks (DSN’02), Washington, DC, 22–26 June 2002. New York: IEEE.

30. Arulraj J, Chang P, Jin G, et al. Production-run software failure diagnosis via hardware performance counters. In: ASPLOS’13, Houston, TX, 16–20 March 2013. New York: ACM.

31. Nakka N, Saggese GP, Kalbarczyk Z, et al. An architectural framework for detecting process hangs/crashes. In: 5th European dependable computing conference on dependable computing (EDCC-5), Budapest, Hungary, 20–22 April 2005. Berlin: Springer.

32. Nakano J, Montesinos P, Gharachorloo K, et al. ReVi/eO: efficient handling of I/O in highly-available rollback-recovery servers. In: The twelfth international symposium on high-performance computer architecture, Austin, TX, 11–15 February 2006. New York: IEEE.

33. Vishwanath KV and Nagappan N. Characterizing cloud computing hardware reliability. In: Proceedings of the 1st ACM symposium on cloud computing (SoCC 2010), Indianapolis, IN, 10–11 June 2010. New York: ACM.

34. Schroeder B, Pinheiro E and Weber WD. DRAM errors in the wild: a large-scale field study. In: Proceedings of the eleventh international joint conference on measurement and modeling of computer systems, SIGMETRICS/performanc 2009, Seattle, WA, 15–19 June 2009. New York: ACM.
35. Wang C, Fu Z, Chen H, et al. Characterizing the effects of intermittent faults on a processor for dependability enhancement strategy. *Sci World J* 2014; 2014: 1–12.

36. Rashid L, Pattabiraman K and Gopalakrishnan S. Characterizing the impact of intermittent hardware faults on programs. *IEEE T Reliab* 2015; 64(1): 297–310.

37. Arasteh B. Improving the resiliency of software against soft-errors without external redundancy and performance overhead. *J Circuit Syst Comp* 2017; 26(7): 1750124.

38. Mueller-Gritschneder D, Dittrich M, Weinzierl J, et al. Performance and accuracy in soft-error resilience evaluation using the multi-level processor simulator ETISS-ML. In: *ICCAD 2018*, San Diego, CA, 5–8 November 2018. New York: IEEE.

39. Gu W, Kalbarczyk Z, Iyer RK, et al. Characterization of Linux Kernel behavior under errors. In: *Proceedings of the 2003 international conference on dependable systems and networks (DSN 2003)*, San Francisco, CA, 22–25 June 2003. New York: IEEE.

40. Chen D, Jacquessilva G, Kalbarczyk Z, et al. Error behavior comparison of multiple computing systems: a case study using Linux on Pentium, Solaris on SPARC, and AIX on POWER. In: *IEEE Pacific rim international symposium on dependable computing (PRDC 2008)*, Taipei, Taiwan, 15–17 December 2008. New York: IEEE.

41. Xiao G, Zheng Z, Yin B, et al. An empirical study of fault triggers in the Linux operating system. *IEEE T Reliab* 2019; 68(4): 1356–1383.

42. Sterpone L and Violante M. An analysis of SEU effects in embedded operating systems for real-time applications. In: *Proceedings of the IEEE international symposium on industrial electronics*, Vigo, 4–7 June 2007, pp.3345–3349. New York: IEEE.

43. Le M and Tamir Y. Fault injection in virtualized systems—challenges and applications. *IEEE T Depend Secure* 2015; 12(3): 284–297.

44. Pham C, Chen D, Kalbarczyk Z, et al. CloudVal: a framework for validation of virtualization environment in cloud infrastructure. In: *IEEE/IFIP international conference on dependable systems & networks*, Hong Kong, China, 27–30 June 2011. New York: IEEE.

45. Feinbube L, Pirl L, Troger P, et al. Software fault injection campaign generation for cloud infrastructures. In: *2017 IEEE international conference on software quality, reliability and security companion (QRS-C)*, Prague, 25–29 July 2017. New York: IEEE.

46. Feinbube L, Pirl L, Troger P, et al. Dependability stress testing of cloud infrastructures. In: *2017 18th international conference on parallel and distributed computing, applications and technologies (PDCAT)*, Taipei, Taiwan, 18–20 December 2017. New York: IEEE.

47. Yamato Y, Nishizawa Y, Nagao S, et al. Fast and reliable restoration method of virtual resources on OpenStack. *IEEE T Cloud Comput* 2015; 6: 572–583.

48. Yuan Y, Li Y, Wu Z, et al. Dependability analysis on OpenStack IaaS cloud: bug analysis and fault injection. In: *IEEE international conference on cloud computing technology & science*, Singapore, 15–18 December 2015. New York: IEEE.

49. Herscheid L, Richter D and Polze A. Experimental assessment of cloud software dependability using fault injection. In: *6th doctoral conference on computing, electrical and industrial systems (DoCEIS)*, Costa de Caparica, 13–15 April 2015. Cham: Springer.

50. Zimba A, Chen H and Wang Z. Bayesian network based weighted APT attack paths modeling in cloud computing. *Future Gener Comp Sy* 2019; 96: 525–537.

51. Wang C, Fu ZC and Cui G. A neural-network-based approach for diagnosing hardware faults in cloud systems. *Adv Mech Eng* 2019; 11(2): 1–10.

52. Cheng E, Mueller-Gritschneder D, Abraham J, et al. Cross-layer resilience, fault tolerance, reliability. In: *Proceedings of the 56th annual design automation conference (DAC 2019)*, Las Vegas, NV, 2–6 June 2019. New York: IEEE.