UNIVERSIDADE D
COIMBRA

Frederico Manuel Duarte Cerveira

Evaluating And Improving Cloud Computing Dependability

Tese no âmbito do Programa de Doutoramento em Ciências e Tecnologias da Informação orientada pelo Professor Doutor Raul André Brajczewski Barbosa e pelo Professor Doutor Henrique Santos do Carmo Madeira e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Maio de 2021

This research has been developed as part of the requirements of the Doctoral Program in Information Science and Technology of the Faculty of Sciences and Technology of the University of Coimbra. This work is within the Dependable Systems specialization domain and was carried out in the Software and Systems Engineering Group of the Center for Informatics and Systems of the University of Coimbra (CISUC).

Funding for this work was provided by the Portuguese Research Agency *Fundação para a Ciência e Tecnologia* (FCT) through the scholarship SFRH/BD/130601/2017.

This work has been supervised by **Professor Raul André Brajczewski Barbosa**, Assistant Professor (Professor Auxiliar), Department of Informatics Engineering, Faculty of Sciences and Technology, University of Coimbra and co-supervised by **Professor Henrique Santos do Carmo Madeira**, Full Professor (Professor Catedrático), Department of Informatics Engineering, Faculty of Sciences and Technology, University of Coimbra.

ABSTRACT

Cloud computing has become the preferred choice by the large majority of organizations to obtain computing resources. Despite the various advantages of cloud computing, it has experienced difficulty in finding adoption by organizations that have mission-critical workloads with strict dependability requirements, as it is at a disadvantage when compared to dedicated infrastructures due to its own *ethos*. Cloud computing consists in the sharing of computing resources through a network to multiple clients that use the same hosted infrastructure. Resource sharing is usually accomplished using virtualization, however both the practice of sharing the same physical resources and the usage of virtualization increase the impact that a failure may have and the likelihood of failure occurrence.

If cloud computing is to be regarded as a trusted platform to support mission-critical workloads, then it must provide similar levels of dependability as those attained by dedicated infrastructures. This thesis addresses the aforementioned issue through the evaluation of the current state of cloud computing dependability and the proposal of contributions that increase its dependability. Both actions are interlinked, since the design of fault tolerance mechanisms that can balance fault coverage and performance overhead requires detailed knowledge about the manner in which cloud computing fails. This information can be extracted from realistic failure data, which is obtained in this thesis using an experimental methodology that employs fault injection for accelerating the data collection process.

To support the experimental campaigns, the ucXception framework and the fault injection tools associated with it have been developed from the ground up as part of this thesis. Before the ucXception framework, various fault injection tools had been developed, but few were capable of being used in the context of cloud computing and virtualization, as well as supporting fault models representative of transient hardware faults and software faults.

The experimental campaigns yield new findings, from which we highlight the observation that faults in the hypervisor and privileged virtual machine can cause common-mode failures that affect multiple clients at once, and, in some cases, lead to silent data corruption. Faults during the execution of guest virtual machines largely cause failures that lead to downtime of the applications in the virtual machine and which cannot propagate to other virtual machines or to the hypervisor, thus suggesting that mature virtualization solutions provide good isolation.

Using this knowledge, we create Romulus, a fault tolerance technique that can tolerate hypervisor failures by migrating virtual machines from the failed hypervisor to a co-located hypervisor. Romulus provides coverage of software and transient hardware faults without requiring redundant hardware and with low downtime, which contributes to its goal of increasing the availability of cloud computing infrastructure. A proof-of-concept implementation is developed and evaluated using fault injection, thereby showing that it can often recover at least part of the virtual machines in a system after a failure.

Furthermore, we propose the Availability-as-a-Service framework for promoting the availability of cloud infrastructure at an agreeable performance cost. The framework uses nested virtualization to host a minimal microvisor that contains just the core logic and depends on modules that encompass specific mechanisms that provide fault tolerance and which can be enabled and disabled explicitly by the cloud provider and client.

Keywords: Cloud Computing, Virtualization, Dependability, Availability, Fault Injection, Fault Tolerance.

RESUMO

A computação em nuvem tornou-se na escolha predileta de uma grande maioria de organizações no momento de adquirir recursos computacionais. Apesar das múltiplas vantagens da computação em nuvem, regista-se dificuldade em que esta seja adotada em cenários nos quais existam requisitos estritos de confiabilidade. O próprio *ethos* da computação em nuvem coloca-a numa posição desfavorável quando comparada com uma infraestrutura dedicada. A computação em nuvem é a partilha de recursos computacionais para vários clientes através de uma rede, sendo normalmente implementada usando virtualização. Todavia tanto a prática de partilhar os mesmos recursos físicos como o uso de virtualização aumentam o impacto que uma avaria pode ter e a probabilidade da sua ocorrência.

Para que a computação em nuvem possa ser considerada como uma plataforma adequada e capaz de suportar cargas de trabalho críticas, esta deve providenciar níveis de confiabilidade semelhantes aos de infraestruturas dedicadas. A tese aborda este problema através da avaliação do estado da arte da confiabilidade da computação em nuvem e da criação de propostas para aumentar a sua confiabilidade. Ambos estes aspetos estão relacionados, pois o desenho de mecanismos de tolerância a falhas capazes de obter um bom equilíbrio entre cobertura de falhas e custo de desempenho implica um conhecimento detalhado sobre os vários tipos de avarias que afetam a computação em nuvem. Por sua vez, essa informação pode ser extraída de dados de avarias realísticos, que são obtidos nesta tese com recurso a uma metodologia experimental baseada no uso de injeção de falhas para acelerar o processo de coleção de dados.

Para dar suporte às campanhas experimentais, a *framework ucXception* e as ferramentas de injeção de falhas que lhe estão associadas foram desenvolvidas de raiz durante o decurso deste doutoramento. A *framework ucXception* é construída a partir de vários anos de estado da arte em ferramentas de injeção de falhas, mas adiciona suporte para modelos de falhas representativos de falhas transitórias de *hardware* e falhas de *software*, para além de ser orientada para o uso em contexto de sistemas virtualizados e de computação em nuvem.

As campanhas experimentais resultaram em várias descobertas, das quais salientamos a observação de que falhas no *hypervisor* e na máquina virtual privilegiada podem levar a avarias que afetam vários clientes simultaneamente e com menor frequência podem potenciar a corrupção silenciosa de dados. Em regra, falhas durante a execução das máquinas virtuais hóspedes levam a avarias que causam indisponibilidade das aplicações que executam na própria máquina virtual e

que não é propagada para outras máquinas virtuais ou para o *hypervisor*, o que sugere que soluções de virtualização maduras providenciam bom isolamento.

Usando esta informação, criámos a técnica de tolerância a falhas denominada por Romulus, que é capaz de tolerar avarias do *hypervisor* através da migração das máquinas virtuais do *hypervisor* avariado para um *hypervisor* localizado no mesmo sistema físico. O Romulus consegue cobrir falhas de *software* e falhas transitórias de *hardware* sem necessitar de *hardware* redundante e obtendo um baixo tempo de inatividade, factos que contribuem para alcançar o objetivo de melhorar a disponibilidade da infraestrutura que suporta a computação em nuvem. Uma prova de conceito foi desenvolvida e avaliada através da injeção de falhas, assim demonstrando que frequentemente é possível recuperar parte das máquinas virtuais do sistema após avaria.

Para além disso, propusemos a *framework Availability-as-a-Service*, cujo objetivo é melhorar a disponibilidade da infraestrutura de computação em nuvem sem um custo de performance exagerado. A *framework* usa virtualização *nested* para alojar um pequeno *microvisor* que trata apenas da lógica essencial e que depende de módulos que implementam mecanismos de tolerância a falhas e que podem ser ligados e desligados explicitamente pelos provedores e clientes da computação em nuvem.

Palavras-chave: Computação em Nuvem, Virtualização, Confiabilidade, Disponibilidade, Injeção de Falhas, Tolerância a Falhas.

PUBLICATIONS

This thesis is partly based on the published scientific research presented in the following peer reviewed papers:

I Cerveira, F., Barbosa, R., & Madeira, H. (2017, January). Soft errors susceptibility of virtualization servers. In 2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC) (pp. 125-134). IEEE.

II Cerveira, F., Barbosa, R., Mercier, M., & Madeira, H. (2017, September). On the emulation of vulnerabilities through software fault injection. In 2017 13th European dependable computing conference (EDCC) (pp. 73-78). IEEE.

III Cerveira, F., Barbosa, R., & Madeira, H. (2017, October). Experience report: On the impact of software faults in the privileged virtual machine. In 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE) (pp. 136-145). IEEE.

IV Fonseca, A., Cerveira, F., Cabral, B., & Barbosa, R. (2017, November). Language-based expression of reliability and parallelism for low-power computing. IEEE Transactions on Sustainable Computing, 3(3), 153-166.

V Cerveira, F., Kocsis, I., Barbosa, R., Madeira, H., & Pataricza, A. (2018, July). Exploratory Data Analysis of Fault Injection Campaigns. In 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS) (pp. 191-202). IEEE.

VI Cerveira, F., Fonseca, A., Barbosa, R., & Madeira, H. (2018, September). Evaluating the inherent sensitivity of programming languages to soft errors. In 2018 14th European Dependable Computing Conference (EDCC) (pp. 65-72). IEEE.

VII Cerveira, F., Fonseca, A., Barbosa, R., & Madeira, H. S. (2019, January). Soft error sensitivity and vulnerability of languages and their implementations. International Journal of Critical Computer-Based Systems, 9(4), 318-347.

VIII Barbosa, R., Cerveira, F., Gonçalo, L., & Madeira, H. (2019, August). Emulating representative software vulnerabilities using field data. Computing, 101(2), 119-138.

IX Cerveira, F., Barbosa, R., & Madeira, H. (2019, September). Fast local VM migration against hypervisor corruption. In 2019 15th

European Dependable Computing Conference (EDCC) (pp. 97-102). IEEE.

X Cerveira, F., Barbosa, R., Madeira, H., & Araújo, F. (2020, February). The Effects of Soft Errors and Mitigation Strategies for Virtualization Servers. IEEE Transactions on Cloud Computing.

XI Cerveira, F., Oliveira, R. A., Barbosa, R., & Madeira, H. (2020, October). Evaluation of RESTful frameworks under soft errors. In 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE) (pp. 369-379). IEEE.

Furthermore, this thesis is also based in two other works scientific research works that have been submited and are awaiting review:

XII Cerveira, F., Barbosa, R., & Madeira, H. ucXception: A framework for evaluating dependability of software systems. (Submitted to EDCC 2021)

XIII Cerveira, F., Barbosa, R., & Madeira, H. Mitigating virtualization failures through migration to a co-located hypervisor. (Submitted to IEEE Access)

ACKNOWLEDGEMENTS

I would like to begin by thanking my parents, grandparents and remaining members of my family for all the support during my academic life and, in particular, during the course of this doctorate. Without their help I would not have been able to reach where I am today.

I would like to greatly thank my supervisor, Prof. Raul Barbosa, for his tireless effort in guiding me during this doctorate and the tremendous help provided during the writing of this thesis. I have learned a lot from you.

I would also like to thank my co-supervisor, Prof. Henrique Madeira, for his insightful opinions and provoking ideas, which have served as the basis for some of the research developed during my doctoral stint.

I would like to thank my friends and colleagues at the Software and Systems Engineering group of the Centre of Informatics and Systems of the University of Coimbra, including, in no particular order, Nádia Medeiros, Charles Gonçalves, Gonçalo Carvalho, João Campos, Jomar Domingos, and many others for all the interesting discussions and mutual support in our effort to produce quality research and obtain a doctoral degree. I hope you find the strength needed to successfully finish this challenge!

I would like to thank two other former colleagues, friends and co-authors of mine, which since the time we first met have successfully obtained their doctoral degree and proceeded with their life. Despite not being physically nearby, they have remained in contact, provided their support whenever necessary and fomented our friendship and professional relations. To Rui Oliveira and Alcides Fonseca, thank you for everything and hope that we have the chance to meet again many more times!

I would like to thank my friends that have helped to distract my mind from work and to make these years more enjoyable. To André Martins, Guilherme Franco, Alexandre Pinto, Seyma Soydemir, and Ivano Elia, a big thank you for your friendship and support!

To everyone else that has helped me during this time, thank you very much. It will not be forgotten!

CONTENTS

I CLOUD COMPUTING AND DEPENDABILITY 1
1 INTRODUCTION 3
1.1 Contributions 5
1.2 Structure of the document 6
2 WHY DOES THE CLOUD FAIL AND WHAT IS BEING DONE ABOUT IT? 9
2.1 Background 10
2.1.1 Cloud Computing 11
2.1.2 Virtualization 13
2.1.3 Dependability 17
2.2 Threats 19
2.2.1 Software Faults 21
2.2.2 Operator Faults 22
2.2.3 Hardware Faults 23
2.3 Means of attaining dependability 25
2.4 Summary 28
II EVALUATING CLOUD COMPUTING DEPENDABILITY 31
3 EVALUATING DEPENDABILITY OF CLOUD COMPUTING 33
3.1 Background in Fault Injection 34
3.2 The ucXception fault injection framework 35
3.2.1 Components of the ucXception framework . . . 36
3.2.2 Fault injection tools of the ucXception framework 40
3.3 Related work 46
3.4 Summary 48
4 ON SOFT ERRORS AND CLOUD SYSTEMS 51
4.1 Methodology 52
4.1.1 Experimental setup 52
4.1.2 Workload 53
4.1.3 Fault model 54
4.1.4 Fault injection tool 54
4.2 Failure modes and classification 55
4.3 Soft errors occurring inside a VM 58
4.3.1 HTTP workload 59
4.3.2 TPC-VMS workload 63
4.4 Soft errors occurring inside the PVM 66
4.5 Soft errors occurring in the hypervisor 69
4.5.1 In the hypervisor memory 69
4.5.2 During the execution of hypercalls 70
4.6 Related work 76
4.7 Summary 76
5 ON SOFTWARE FAULTS AND CLOUD SYSTEMS 79

5.1 Methodology . . . 80
5.1.1 Architecture & target components . . . 80
5.1.2 Workload . . . 83
5.1.3 Fault injection technique & faultload . . . 84
5.2 Software faults in the toolstack . . . 85
5.3 Software faults in device drivers . . . 87
5.4 Related work . . . 87
5.5 Summary . . . 88

III IMPROVING CLOUD COMPUTING DEPENDABILITY 91

6 AVAILABILITY THROUGH MIGRATION TO A CO-LOCATED HYPERVISOR 93
6.1 Approach . . . 94
6.1.1 Architecture . . . 94
6.1.2 Lifecycle . . . 96
6.1.3 Optimized state extraction . . . 98
6.2 Proof-of-concept . . . 99
6.2.1 Modifications . . . 100
6.2.2 Lifecycle and flow . . . 100
6.2.3 Limitations . . . 104
6.3 Methodology . . . 105
6.3.1 Physical setup . . . 105
6.3.2 Workload and profiles . . . 106
6.3.3 Fault injection . . . 106
6.3.4 Triggering mechanism for the recovery action . 108
6.3.5 Correctness verification . . . 109
6.3.6 Recovery assessment . . . 109
6.4 Evaluation . . . 110
6.4.1 What happens to VMs when the hypervisor fails? 110
6.4.2 Can Romulus provide tolerance against hypervisor failures? . . . 111
6.4.3 How much downtime is incurred during the migration process? . . . 115
6.4.4 How much performance overhead does Romulus introduce? . . . 116
6.5 Limitations . . . 117
6.6 Related Work . . . 118
6.7 Summary . . . 119

7 AVAILABILITY-AS-A-SERVICE 121
7.1 Architecture . . . 123
7.2 Failure Mode Assumption . . . 125
7.3 Hardening the AaaS framework . . . 127
7.4 Modules . . . 129
7.5 Service Interface . . . 131
7.5.1 Microvisor Service Interface . . . 131
7.5.2 Module Service Interface . . . 133

7.6 Limitations 133
7.7 Case Study: Infrastructure reboot using an external watchdog 134
7.8 Case Study: Integrating Romulus into AaaS 136
7.8.1 Service Interface 136
7.8.2 Manifest 137
7.9 Related Work 138
7.10 Summary 139
8 conclusion 141

bibliography 145

LIST OF FIGURES

Figure 1 Types of hypervisors. 15
Figure 2 Simplified architecture of the Xen hypervisor. . 16
Figure 3 Fault-error-failure causality chain. 19
Figure 4 Example of configuring fault injection locally. . 36
Figure 5 Example of configuring fault injection in a remote node. 36
Figure 6 Architecture of the ucXception framework. . . 37
Figure 7 Flow of the fault injection tool for Linux-based systems. 42
Figure 8 Flow of the fault injection tool for virtualized systems. 43
Figure 9 Flow of the fault injection tool of software faults. 45
Figure 10 Experimental setup. 52
Figure 11 Assembly payload. 56
Figure 12 Manifestation latency, for effective errors, both in PV and HVM, HTTP workload. 62
Figure 13 Number of client requests affected by a single bit-flip error, for HTTP workload. 62
Figure 14 Distribution of failure modes across processor registers, for injections in application processes and HTTP workload. 64
Figure 15 Failure percentage in processes of the PVM. . . 69
Figure 16 Heatmap of failure probability for each register/fault location pair. 73
Figure 17 Architecture of a virtualized system, focusing on the PVM. 81
Figure 18 Model of the experimental setup. 83
Figure 19 Failure modes and probabilities when injecting on the toolstack. 86
Figure 20 Failure modes and probabilities when injecting on a device driver. 87
Figure 21 Architecture and layers of Romulus. 95
Figure 22 Code added to nvmx_n2_vmexit_handler. . . . 101
Figure 23 Code of the iterate_ept_structures function that was added to Xen. 102
Figure 24 Code of the update_ept_for_new_host function that was added to Xen. 103
Figure 25 Lifecycle and actions. 104
Figure 26 Experimental setup used for the experiments. 105
Figure 27 Resource usage of the workload and its profiles. 107
Figure 28 Total recovered VMs after a hypervisor failure. 113

Figure 29 Cumulative histogram of recovery probability. 114
Figure 30 Downtime in a single VM setup. 115
Figure 31 Duration of the recovery process per VM size. 116
Figure 32 Architecture of the Availability-as-a-Service framework. 124

LIST OF TABLES

Table 1 Summary of threats to the dependability of cloud computing and failure modes that they cause. 29
Table 2 Fault injection tools of ucXception. 41
Table 3 Software fault model operators. 44
Table 4 Outcomes of fault injection in application processes within a VM, HTTP workload. 59
Table 5 Outcomes of fault injection in OS processes within a VM, HTTP workload. 61
Table 6 Outcomes of fault injection in application processes within a VM, TPC-VMS workload. . . . 65
Table 7 Outcomes of fault injection in OS processes within a VM, TPC-VMS workload. 66
Table 8 Outcomes of fault injection in PVM, HTTP workload. 67
Table 9 Outcomes of fault injection in PVM, TPC-VMS workload. 68
Table 10 Hypercall profiling for both workloads. 70
Table 11 Outcomes of faults affecting the iret hypercall of Xen, HTTP workload. 71
Table 12 Outcomes of faults affecting the stack_switch hypercall of Xen, HTTP workload. 71
Table 13 Outcomes of faults affecting the iret hypercall of Xen, TPC-VMS workload. 72
Table 14 Outcomes of faults affecting the stack_switch hypercall, TPC-VMS workload. 72
Table 15 Software metrics of the target components. . . 82
Table 16 Statistical analysis of both workload profiles. . 107
Table 17 Recovery probability (1 VM). 111
Table 18 Fault injection statistics. 112
Table 19 Performance overhead of Romulus compared against different setups. 117
Table 20 Failure modes of a cloud computing node. . . 126
Table 21 Association between failure modes and modules.131
Table 22 Essential services of the microvisor internal service interface. 132
Table 23 Example of the microvisor external service interface. 133
Table 24 Example of the module service interface. . . . 133
Table 25 Evaluation of the external watchdog and reset. 136

Table 26 Service interface of a module that implements Romulus. 136

ACRONYMS

AFR Annualized Failure Rate

CPU Central Processing Unit

CSV Comma-Separated Values

DIMM Dual In-line Memory Module

DRAM Dynamic Random Access Memory

ECC Error Correcting Code

EPTP Extended Page Table Pointer

EPT Extended Page Tables

FIT Failures in Time

FI Fault Injection

FPU Floating Point Unit

HDD Hard disk drive

HVM Hardware Virtual Machine

HWIFI Hardware-Implemented Fault Injection

IaaS Infrastructure-as-a-Service

LoC Lines of Code

MTBF mean time between failures

MTTR Mean Time To Repair

NFS Network File System

OS Operating System

PaaS Platform-as-a-Service

PVM Privileged Virtual Machine

SaaS Service-as-a-Service

SLA Service-level Agreement

SLO Service-level Objective

SSE Streaming SIMD Extensions

SUT System Under Test

SWIFI Software-Implemented Fault Injection

TLB Translation Lookaside Buffer

vCPU Virtual CPU

VMCB Virtual Machine Control Block

VMCS Virtual Machine Control Structure

VM Virtual Machine

Part I

CLOUD COMPUTING AND DEPENDABILITY

1 INTRODUCTION

Cloud computing pervades society as we known it today, even if the abstract nature of the "cloud" hides this constatation. Cloud computing is a paradigm whereby computing resources are accessed through a network and hosted in an infrastructure that is rented out to multiple clients. The large majority of the services available through the Internet are supported by cloud computing, be it the social platforms that we use to communicate with others, the news platforms, the giant e-commerce retailers or many other less publicized services.

Organizations and companies have flocked to the cloud because of the many advantages that it offers, namely lower monetary expenses due to non-existent acquisition costs and paying only for the resources that are used, as well the ability to automatically scale the contracted resources according to load and demand. Scalability, elasticity and resource sharing are pillars of cloud computing that have proven to represent a large change from traditional deployment models where workloads were assigned to dedicated hardware, which would be underutilized during large portions of the time and unable to follow large spikes in load.

Resource sharing is attained through virtualization, an enabling technology of cloud computing that has long existed but only reached critical mass around the year 2000, when the first consumer microprocessors capable of fully virtualizing the x86 architecture were produced. Since then, virtualization has very much contributed to the adoption of cloud computing, which has trended upwards since around 2006. Virtualization is a technique that presents a virtual copy of the physical hardware to the software applications and operating system, thereby giving the impression that the entire machine is exclusively allocated to them, when in reality multiple applications hosted in different virtual machines are being executed.

This thesis deals with one of the reasons why cloud computing is not more widely adopted, specifically by organizations possessing mission-critical workloads, which is the dependability of cloud computing. Despite its enticing advantages, cloud computing has yet to find adoption from all potential clients. Organizations that rely on workloads with strict dependability requirements, namely in terms of availability and reliability, are reticent to migrate their workloads to the cloud. These workloads can support use cases such as telecare, home banking, brokers and other healthcare and financial activities. For these use cases, organizations prefer to keep workloads hosted in dedicated infrastructure over which they maintain full control, in

part due to the lack of guarantees that cloud computing is capable of ensuring.

Public cloud computing providers offer service-level agreements (SLAs) which specify, among others, the expected availability. Current agreements available to the public indicate that even the most popular cloud providers do not offer availability levels higher than 99.99% (*i.e.*, more or less 4 minutes of downtime in a month), which is insufficient for workloads that demand high-availability. Although SLAs negotiated between large organizations and public or private cloud providers may offer higher levels of availability, these trail what can be accomplished using specialized, dedicated infrastructure. When looking at reliability of cloud computing, the situation is even more precarious since SLAs completely disregard this non-functional attribute due to the difficulty in accurately and fairly measuring reliability, paired with a lack of interest from cloud providers and demand from cloud clients.

Although virtualization opens a number of possibilities for cloud computing, it also exposes the cloud to new risks. For example, consolidation of workloads over the same physical hardware is a tenet of cloud computing that is accomplished thanks to virtualization. However the higher the amount of consolidation, the higher the stress on the system and the impact that a failure may have, since it can affect more users at once. Another example of why virtualization leads to higher exposure to dependability threats is the requirement for a bigger software stack that is needed for providing virtualization (*e.g.*, the hypervisor, privileged virtual machine and toolstack) and managing the complexity of cloud computing (*e.g.*, cloud management software like OpenStack), which is expected to introduce software faults in the system. Finally, the software needed to provide virtualization (*e.g.*, the hypervisor) effectively constitutes a single point-of-failure that can affect the entire machine and all of its clients at once.

In order to close the gap between the dependability attainable using dedicated infrastructure and that of cloud computing, thus reducing a major obstacle to true widespread adoption and migration of mission-critical workloads to the cloud, the goals of this thesis are i) *to evaluate cloud computing dependability* and ii) *to propose contributions that improve cloud computing dependability*. An approach based on the experimental method was used to collect information needed to design adequate fault tolerance mechanisms. More specifically, fault injection was employed to accelerate the collection of failure data that is required to evaluate how the virtualized infrastructure that supports the cloud behaves when affected by transient hardware faults and software faults. Throughout this thesis, cloud computing dependability is analyzed at the infrastructure level with a focus in the node that resorts to virtualization for sharing its computing resources. Although other levels, such as the application-level or the distributed-level, also have a role on the dependability of cloud computing as experienced by the

cloud clients, we focused in the infrastructure because it is there that the current body of knowledge is less developed and cloud computing introduces most of its innovations.

The following sections describe the key contributions of this thesis in detail and explain the structure of this document.

1.1 CONTRIBUTIONS

The following list details the main contributions found in this thesis:

1. *A review of the state-of-the-art on the topic of cloud computing dependability* – which identifies the characteristics that make cloud computing unique, justifies why cloud computing warrants a study focusing on its dependability, lists the threats that affect cloud computing dependability, as well as the mechanisms and techniques that are used to provide fault tolerance in the cloud (presented in Chapter 2).

2. *A framework for evaluating cloud computing and virtualized systems using fault injection* – which is one of the few publicly available fault injection projects supporting injection of various fault models (namely, transient hardware faults and software faults) and injection in virtualized and cloud computing systems. It was developed during this doctorate to support the experimental campaigns that were used to evaluate cloud computing dependability and validate the proposed contributions (presented in Chapter 3, based on Publication XII).

3. *An evaluation of the impact of transient hardware faults in a virtualized system* – which provides novel insights and observations regarding how a virtualized system that supports cloud computing deployments behaves when affected by transient hardware faults. This analysis aims to characterize the various failure modes that occur in a virtualized infrastructure, as well as verifying the error isolation afforded by a mature hypervisor (presented in Chapter 4, based on Publications I and X).

4. *An evaluation of the impact of software faults in a virtualized system* – which provides new insights into the impact that software faults in the components that compose a traditional virtualized system may have (presented in Chapter 5, based on Publication III).

5. *A fault tolerance mechanism to tolerate hypervisor failures* – which provides tolerance without requiring external hardware and in a transparent manner. It employs efficient migration of VM state across two hypervisors hosted in the same physical machine to enable the VMs to continue executing if their hypervisor fails and without requiring a reboot. The proposed technique

is complemented with a publicly available proof-of-concept implementation that has been validated and evaluated using fault injection (presented in Chapter 6, based on Publication XIII).

6. *A framework for increasing the availability of cloud computing infrastructure* – which exposes a service interface to cloud providers and clients that can be used to configure a set of modules that provide availability. These modules integrate into the framework, which was designed according to the knowledge and experience obtained during the course of this work (presented in Chapter 7).

1.2 STRUCTURE OF THE DOCUMENT

Due to the different objectives tackled in this thesis, we decided to organize it into three distinct parts, namely:

- Part I: Cloud Computing and Dependability
- Part II: Evaluating Cloud Computing Dependability
- Part III: Improving Cloud Computing Dependability

Part I encompasses the current chapter and Chapter 2, which presents the state-of-the-art in cloud computing dependability and fault tolerance, as well as presenting the essential concepts of cloud computing, virtualization and dependability.

Part II describes the work performed to evaluate cloud computing dependability and is composed by Chapters 3, 4 and 5. Chapter 3 familiarizes the reader with fault injection and describes the ucXception framework, which was developed as part of this thesis for supporting the required fault injection campaigns. Chapter 4 contains the results of the evaluation of the impact that soft errors (*i.e.*, transient hardware faults) can have in a virtualized infrastructure. Chapter 5 analyses the effect that software faults (*i.e*, software bugs) in components of the privileged virtual machine can have in a virtualized infrastructure.

Part III presents two contributions to improve cloud computing dependability. Chapter 6 presents Romulus, a fault tolerance technique for attaining high-availability operation even in the presence of hypervisor failures. Furthermore, a proof-of-concept implementation of Romulus was developed over the source code of the Xen hypervisor and evaluated through fault injection of transient hardware faults and software faults in the hypervisor. Chapter 7 presents the Availability-as-a-Service framework, which has the objective of increasing the availability of cloud computing while maintaining a lean performance overhead that is configurable by cloud providers and clients through a service interface. Finally, Chapter 8 presents the conclusion of the thesis, where we reflect on the work that was performed, how this thesis contributes to a better understanding and improvement of cloud computing dependability and possible future work.

2 WHY DOES THE CLOUD FAIL AND WHAT IS BEING DONE ABOUT IT?

Since the origin of computer systems, there has been a constant refinement and improvement in approaches to optimize the available resources and expand the boundaries of what can be accomplished, which has lead to the appearance of trends such as distributed systems, relational databases and cloud computing, to name a few. Nevertheless, the need for dependable and highly available systems is timeless and implies understanding *why computers fail* and *what can be done about it*. To this purpose, Jim Gray authored one of the pioneering studies [58] that analyzed these exact questions with respect to the Tandem line of fault-tolerant computers, which was reproduced by Oppenheimer *et al.* some decades later after authoring a paper that focused on Internet services, the hot and rapidly growing topic at the time, and analyzed how they fail and what should be done to avoid that [101]. Since then, cloud computing has gained widespread adoption and became the standard venue for organizations and individuals to deploy their systems. This chapter follows the footsteps of its predecessors by providing the answers to *why cloud computing fails and what is being done about it*.

These two questions must be answered specifically for cloud computing because it differs from any other trend or technology that has come before and possesses unique characteristics that create new challenges to ensuring dependability. One such challenge is related to the technology that is more often associated with cloud computing, which is virtualization. Virtualization is employed in the large majority of cloud computing deployments with the purpose of enabling workload consolidation over a shared physical machine, while ensuring isolation between the tenants (*i.e*, the different users of the machine, which are usually the cloud clients). Although virtualization is nowadays sufficiently mature to be trusted by most cloud providers and clients, specifically if the workload is not critical, it implies the addition of many thousands of lines of software code in the form of the hypervisor and other auxiliary applications, toolstack and device drivers, which will inevitably introduce software faults (*i.e.*, software defects or bugs) to a layer that represents a single point-of-failure.

Another challenge unique to cloud computing is the consolidation of multiple workloads (or tenants, cloud clients, VMs, *etc.*) over the same hardware using virtualization. By consolidating more than one client, the potential impact of a failure is largely amplified, as it is multiplied by all the clients that are executing over the hardware. Moreover

consolidation increases the average usage level of the system, which tends to increase the rate at which failures occur [20, 52].

Other challenges appear because of the characteristics of the datacenters that support cloud computing, specifically the datacenters of large public cloud providers. These datacenters contain a tremendous amount of hardware that is connected through high speed networks and are often referred to as hyperscale infrastructure due to supporting systems with very high degree of scalability, such as large public cloud computing providers. Given their size and the raw amount of physical components, hardware failures are a daily occurrence [52]. Furthermore, these datacenters are extremely complex to manage, thus requiring the usage of cloud management software. As such, complexity and extra software lead to the appearance of unexpected failures that are not anticipated during the design and development phase of a cloud computing infrastructure [53] and to *gray failures*, which leave the system in a degraded mode because they remain undetected up until the moment when the entire system abruptly fails [67]. Finally, the cost-sensitive and very competitive nature of the business model of cloud providers has sustained the popularization of energy saving techniques, such as dynamic frequency and voltage scaling, which have been shown to significantly increase the failure rate of hardware [26].

All of the enumerated challenges are in one way or another unique to cloud computing and must be handled using adequate approaches. Thankfully the nature of cloud computing also creates opportunities that can be used to tackle these challenges. For example, the large distributed nature and existence of large amount of hardware, most of which is underused during large parts of the day, makes it a favorable proposition to migrate or replicate VMs across different hardware or datacenters. Whereas the omnipresence of virtualization leads to the development of creative techniques that provide a degree of redundancy and fault tolerance using the isolation and abstraction of virtualization.

This chapter starts by introducing essential concepts and the background in cloud computing, virtualization and dependability, which are topics that are present throughout the rest of this thesis. Then, it describes the state-of-the-art regarding the threats that risk damaging the dependability of cloud computing and the means to attain dependability that are being used nowadays to maximize dependability in cloud computing.

2.1 BACKGROUND

With the appearance of cloud computing, a range of terms and vocabulary characteristic of virtualization has appeared and entered everyday usage by IT professionals. The same can be said about the

terms associated with the large field of dependability. Sometimes a term definition collides with a meaning assumed in another field, thus confusion in the terminology may arise. Throughout this thesis, concepts and terms in the areas of cloud computing, virtualization and dependability are regularly used, hence this section serves as a compendium that the reader should take advantage of to better comprehend this manuscript.

2.1.1 *Cloud Computing*

Cloud computing has been defined as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (...) that can be rapidly provisioned and released" [92] and has been claimed to "provide on-demand resources and services over a network, usually the Internet, with the scale and reliability of a data center" [59]. Its appearance represents an opportunity for clients to have tremendous amount of computing power at their disposition, which can be quickly allocated and released according to their needs, thus following demand trends and freeing the clients from having to pay for unused resources, as was the case with the traditional on-premises and in-house managed infrastructure. For cloud providers, many of which are big tech giants, cloud computing provides the opportunity to maximize the utilization of their existing systems by selling idle computing power.

Cloud computing can be classified according to three properties [92], which are: i) its characteristics, ii) its service models, and iii) its deployment models.

The essential characteristics of cloud computing are:

- *On-demand self-service* – a customer must be able to provision the desired computing resources automatically and without human intervention from the cloud provider;
- *Broad network access* – the capabilities are accessible over the network using standard protocols that can be universally used;
- *Resource pooling* – the computing resources of the cloud provider are kept in a pool from which they will be taken and returned according to consumer demand;
- *Rapid elasticity* – resources can quickly be provisioned and released according to demand, so that highly scalable systems with seemingly infinite resources can be created;
- *Measured service* – metering of resource usage is required to keep track of how the cloud provider's resources are being used and, possibly, for billing purposes.

The service models of cloud computing (*i.e.*, the manner in which the service is provided to the cloud clients) are:

- *Function as a Service (FaaS)* – the consumer publishes functions (*i.e.*, pieces of code) using agreed programming languages and libraries, which are then executed over the cloud infrastructure. The consumer has no control over any of the underlying software stack or hardware infrastructure;
- *Software as a Service (SaaS)* – the consumer is given access to software applications that are hosted in the cloud and cannot manage or access the underlying infrastructure, such as operating systems. Usually the services are accessible through a web-based thin client;
- *Platform as a Service (PaaS)* – the consumer cannot manage or access the underlying infrastructure, but can deploy his own applications to the cloud;
- *Infrastructure as a Service (IaaS)* – the consumer is allowed to provision and manage all of the resources, as well as being able to deploy and execute software applications and operating systems as desired. Nevertheless, the underlying cloud infrastructure is not managed by the consumer.

Deployment models (*i.e.*, the configuration of the environment with regards to where the infrastructure is located and who has control over it) are:

- *Private cloud* – the cloud is used by a single organization, can be managed by the organization or a third-party, and its supporting infrastructure may be on or off-premises;
- *Community cloud* – similar to a private cloud but may be used by a group of organizations that share a common goal;
- *Public cloud* – it is the most common deployment model. The cloud is open to the public and its infrastructure is managed by a cloud provider;
- *Hybrid cloud* – it is a composition of two or more other clouds that may use different deployment models but share an unified technology or standard to allow seamless migrations between them.

Cloud computing is well-known for its usage of service-level agreements (SLAs), *i.e.*, agreements between cloud provider and cloud client that specify various aspects of the service given by the cloud provider (usually referred to as service-level objectives) that dictate the quality of service, availability, and compensation to be awarded to the clients in the case that any of the objectives are violated.

2.1.2 *Virtualization*

The empowering technology behind cloud computing is virtualization, *i.e.*, "a variety of mechanisms and techniques used to decouple the architecture and user-perceived behavior of hardware and software resources from their physical implementation" [48], where users execute their software systems (operating system and applications) inside of virtual machines (VMs), *i.e.*, "efficient, isolated duplicate of the real machine" [110], over the same physical hardware [120]. Popek and Goldberg's article [110] laid out the requirements for virtualizability:

1. *Efficiency* – states that all innocuous instructions (*e.g.*, non privileged instructions) should be executed directly in the hardware. Its purpose is to avoid performance overhead due to virtualization unless when absolutely necessary;

2. *Resource Control* – states that any application being virtualized should not be able to interfere with the system resources. In other words, virtualization must provide isolation between VMs and the hypervisor and among the VMs themselves;

3. *Equivalence* – states that any application being virtualized executes indistinguishably from how it would if virtualization was not used, except in two situations, timing and resource availability. Timing refers to the fact that virtualization implies software that will occasionally intrude during execution of the application and lead it to take longer to execute than its non-virtualized counterpart. Resource availability deals with the fact that resources are finite and the software required for providing virtualization will reduce the overall available resources to a lower amount than in a non-virtualized counterpart.

These requirements are considered an integral part of full virtualizability to this day. However, hybrid virtualization is also a viable and quite popular technique whereby one of the three requirements for full virtualizability is relaxed.

Virtualization far precedes the appearance of cloud computing, having first been used in the 70s to allow time-sharing large and costly mainframes [48, 120]. However, virtualization soon entered a period of disinterest and decreased adoption, in part due to missing hardware support to enable virtualization with acceptable overhead on the popular x86 architecture. In fact, prior to the appearance of hardware extensions that enabled hardware-supported virtualization and propelled virtualization and cloud computing to become a staple in the IT world, several organizations had attempted to reach efficient x86 virtualization using software-only approaches. This is the case of VMWare and Xen, which pioneered the usage of runtime binary translation [1] and paravirtualization [14, 151, 152] for virtualizing x86

systems. With the development of software-based virtualization approaches and the inclusion of virtualization extensions in the majority of consumer microprocessors, virtualization, which had once been a technology available to the few that could afford large mainframes, has become a technology for the masses [120]. In fact, this thesis is only now feasible in part due to the fact that virtualization has only become widespread relatively recently.

Virtualization is implemented through a hypervisor, sometimes also called as a virtual machine monitor, which provides all the functionalities required for providing virtualization and abstracts the physical resources from the VMs [118]. The hypervisor resorts to a range of techniques for implementing virtualization, which are often grouped into virtualization modes. Some techniques can be classified as belonging to software-based virtualization, such as trap-and-emulate, binary translation and paravirtualization, while the techniques that use functionalities provided by hardware with virtualization extensions are referred to as hardware-based virtualization. Recently there have been efforts to create virtualization modes that mix software and hardware based techniques with the intent of providing the best possible performance, however these virtualization modes are still being actively developed and are rarely found in production subsystems.

Initially, the most prevalent virtualization technique was trap-and-emulate [56], which consisted in trapping the execution of privileged operations inside the VMs and emulating their operation in the hypervisor. The biggest limitation of trap-and-emulate is that it cannot be used to virtualize the x86 architecture, because this architecture has various privileged functions that do not trap when called, thus posing isolation and safety problems since the VMs can interfere with the rest of the system. Later, VMWare and others heavily researched binary translation, which was able to virtualize the x86 architecture by interpreting the instructions that are about to be executed and rewriting a subset (namely the privileged instructions) to prevent violations of isolation. Xen [14] was one of the first hypervisors to employ paravirtualization, a virtualization technique that depends on modifications to the operating system inside the VMs to explicitly communicate with the hypervisor. The advantages of binary translation when compared to paravirtualization are that the original source code and binary do not need to be modified and performance optimizations based on avoiding expensive traps can be performed, whereas paravirtualization promises better overall performance, particularly in highly consolidated settings (*e.g.*, hundreds of VMs in the same hardware) [14, 152].

Hardware extensions from the main microprocessor manufacturers have enhanced x86 virtualization by providing more privilege rings, thus allowing a CPU-level separation between hypervisor, operating system and user-space applications, introducing the concept of VM

entry and exit, which represents the transitions of control between the hypervisor and the VMs, specifying which instructions should lead to traps (in the form of VM exits to hypervisor routines) and supporting interrupt and memory virtualization [19, 147].

A particular type of virtualization has recently started to gain interest and adoption as a tool to support more complex systems. This type of virtualization, which is called nested virtualization, consists on having various levels of virtualization, or, in other words, virtualizing a virtualized system recursively [18]. It has experienced recent interest as a technique for executing already virtualized systems in the cloud [18], performing intrusion detection [51], virus scanning [117] or implementing other security and fault tolerance techniques that operate from outside the VMs that they protect.

Hypervisors may be classified according to their type [133], *i.e.*, the position in the system that they occupy and hence the functionalities that they must implement. A bare-metal hypervisor, also known as a Type 1 hypervisor, is a hypervisor that executes directly on top of the hardware and, therefore, must handle various functionalities usually assigned to operating systems, such as memory and CPU management. A hosted hypervisor, or Type 2 hypervisor, executes above an existing operating system, thus does not need to implement the functionalities provided by the operating system, usually at the expense of worse performance. Figure 1 visually represents these two hypervisor types.

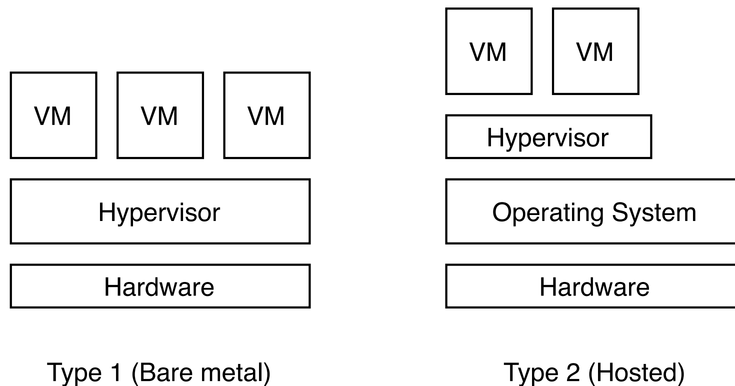


Figure 1: Types of hypervisors.

Many hypervisors, such as Xen, offload certain tasks to privileged virtual machines (PVMs) that execute over it. In Xen the norm is for one PVM, which is called Domain-0 (dom0), to be used to provide the device drivers that will be used by the other (guest) VMs, along with the operating system that the system administrator will use to manage the virtualized system. Although Xen was developed as an academic effort to research paravirtualization, nowadays it has become one of the most mature and popular hypervisors, particularly in cloud computing deployments, and supports other virtualization

modes, including hardware-based modes [63]. The architecture of Xen is presented in a simplified manner in Figure 2, which has been adapted and extended from [14].

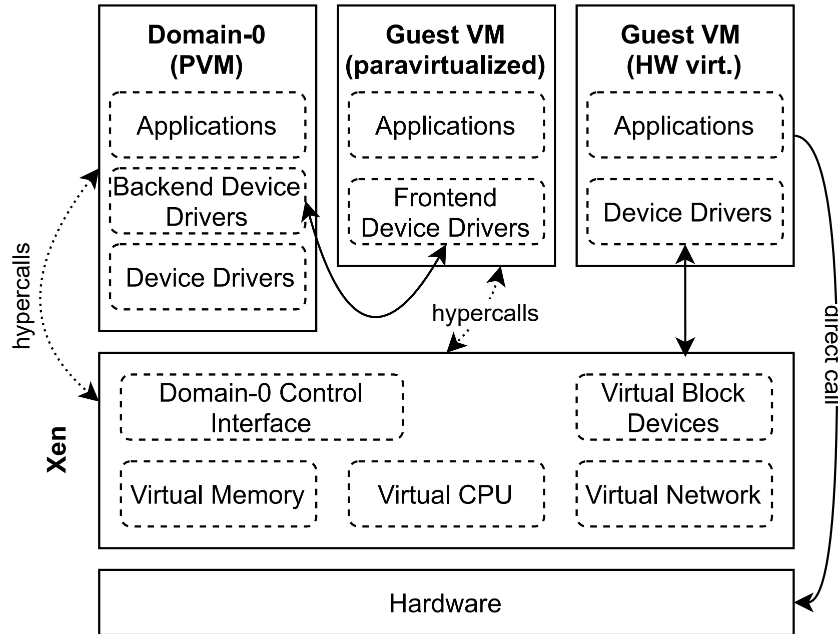


Figure 2: Simplified architecture of the Xen hypervisor.

The hypervisor presents the VMs with virtualized copies of the hardware, including the CPU, the memory, network and other I/O devices, and handles the tasks associated with these, such as scheduling the VMs across the available CPU time. Above the hypervisor, the PVM is needed for providing the device drivers for the hardware that will be used by the guest VMs that use paravirtualization (or hardware-based virtualization extended with paravirtualized device drivers). The device driver model in Xen is divided into backend device drivers, which are located in the PVM and interact with the real device drivers to request operations on the hardware, and frontend device drivers, which are located in the VMs and are used to communicate with the backend drivers. VMs that use pure hardware-virtualization communicate directly with the virtualized hardware using their own device drivers and taking advantage of hardware extensions. Paravirtualized VMs may communicate with the hypervisor, for example to request modifications to page tables, through hypercalls, which are conceptually similar to the system calls of operating systems.

The encapsulated nature of VMs has lead to the popularization of a range of techniques [48], such as suspending and resuming VMs, which increases the portability of systems by allowing entire systems, including operating system and user-space applications, to be moved as a single file. Another popular technique, VM migration, builds upon the suspend/resume functionality to allow runnings VMs to be migrated between different physical machines [32]. Finally, virtual

machine introspection (VMI), *i.e.*, "a technique for viewing the runtime state of a virtual machine" [103], has also been used to monitor and modify the state of the VM directly from the hypervisor without the need for instrumentation inside the VM.

In summary, virtualization represents an enabling technology of cloud computing that carries both increased risks to dependability and opportunities to improve dependability by using fault tolerance techniques that take advantage of virtualization. Given its preponderance in cloud computing, it is a central topic in this thesis.

2.1.3 *Dependability*

Dependability can be defined as "the ability to deliver service that can justifiably be trusted" [7], "the ability to avoid service failures that are more frequent and more severe than is acceptable" [7], "a property of a system that justifies placing one's reliance" [136], or "a property of the system that reflects its trustworthiness" [134]. Dependability is often assumed to encompass five attributes: availability, reliability, safety, integrity and maintainability.

Availability is "readiness for correct service", which reflects the on-demand probability that the service will offer correct service. It is often measured using a number that represents the steady-state availability, *i.e*, the percentage of time that the system is available assuming an operation of infinite duration. This number is often associated to the notion of *X-nines* that represent the amount of nines in the steady-state availability (*e.g.*, five nines means an availability of 99.999%). Another closely related notion is downtime and its inverse, uptime, which refer to the amount of time that the system is in a incorrect (non-working) or working state. For example, an availability value of five nines translates to a downtime lower than 5.26 minutes per year. Availability can be mathematically defined as in Equation 1.

$$\text{Availability} = \frac{\text{MTBF}}{(\text{MTBF} + \text{MTTR})} \tag{1}$$

Where MTBF corresponds to mean-time-between-failures, *i.e.*, how much time is spent between the occurrence of failures of the system, and MTTR refers to mean-time-to-repair, *i.e.*, how long does a failure take to repair. MTBF can be calculated as the arithmetic mean between failures of a existing system and is sometimes described using failures-in-time (FIT), which corresponds to how many failures occur in a billion hours of operation and which can be defined as shown in Equation 2.

$$\text{FIT} = \frac{1}{\text{MTBF}} * 10^9 \tag{2}$$

Mean-time-to-failure (MTTF) is conceptually similar to MTBF but should only be used in systems where repairs are not possible (the

failed component is discarded after failure). Annualized failure rate (AFR) is another concept that can be used to describe the dependability of a component or system and which is defined as the probability that the component will fail during a full year of use. It can be obtained from the MTBF and the total number of hours of operation in an year, as shown in Equation 3.

$$\text{AFR} = 1 - e^{\left(\frac{-(365.2*24)}{\text{MTBF}}\right)} \tag{3}$$

Availability is widely used in cloud computing to measure the quality of the service provided by the cloud provider and is often one of the service-level objectives (SLOs) found in the service-level agreements (SLAs) that are celebrated between cloud providers and clients to describe the expectations in the contracted service.

Reliability refers to the "continuous and correct delivery of a service". It is far less often employed as a metric of quality in cloud computing, despite being equally, if not more, important than availability. However the challenges inherent in measuring reliability have limited its popularity in cloud computing. Safety is the "absence of catastrophic consequences on the user(s) and the environment". Integrity is the "absence of improper system alterations". Maintainability is the "ability to undergo modifications and repairs".

Dependability may be affected whenever its threats – failures, errors and faults – are present in the system. Failures are events that occur when the provided service deviates from the correct service. A failure represents the transition from correct to incorrect state and may have a temporal duration associated to it, which is the service outage. A failure ends when a service restoration takes place. The specific manner in which the system fails is known as the failure mode, which is usually a crash failure mode, a hang failure mode or a failure mode where silent data corruption occurs. Errors are the deviations in a system state that lead to the failures and faults are the root causes of errors.

Faults may be classified regarding their manifestation as permanent (when the fault is always present), transient (when the fault exists during some time but then disappears) or intermittent (when the fault appears and disappears seemingly randomly). Permanent faults are often caused by hardware that has failed, such as a disk or network switch that stopped working completely. Transient faults may be caused by hardware that is failing but is still operating at an unstable state or software that has software defects (bugs) that are activated under specific conditions. Intermittent faults often have harder to discover root causes.

From this description it is clear that there is a causality chain between these three concepts: a fault may be activated and cause an error, which may then lead to a failure. As depicted in Figure 3, an error

may propagate through various parts and components of a system before it reaches the system boundary and becomes a failure. This chain can be extended across multiple connected systems, so that a failure of a system may be a fault to another system that depends on the first.

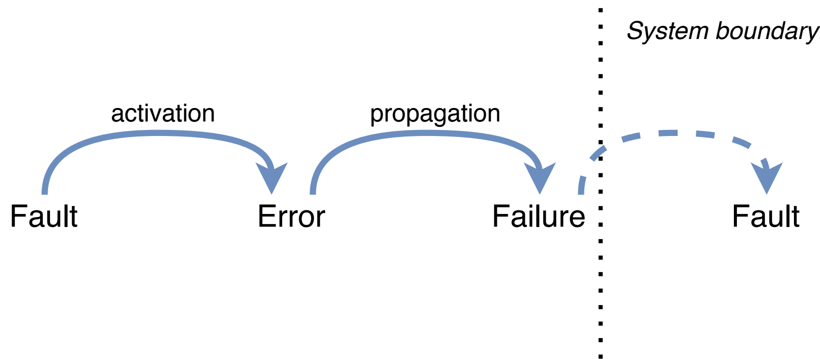


Figure 3: Fault-error-failure causality chain.

In order to deal with the threats to dependability, means to attain dependability must be employed. These means consist of fault prevention, fault tolerance, fault removal and fault forecasting. Fault prevention is the process of ensuring that faults are not introduced to the system, such as the usage of software testing during development to avoid introducing software faults. Fault tolerance aims to avoid failures by carrying out error detection and recovery actions. Error detection employs techniques such as duplicate execution to detect the presence of an error in the system before it causes a failure. Recovery actions depend on the system and faults being considered, but may consist on replicating execution, rolling the state back to a error-free copy (*e.g.*, checkpointing with rollback [76]) or rolling the state forward to a error-free state [112] (*e.g.*, reapplying changes stored in a log to a fresh instance). Fault removal consists in finding and removing faults in the system during development or after deployment, such as performing software testing to find and then fix software faults or applying patch fixes to a running system. Fault forecasting consists in performing an evaluation of the system behaviour with regards to fault activation and occurrence with the objective of modeling the system dependability, identifying and ranking the failure modes or event combinations that may lead to failures, and obtaining probabilistic dependability measures.

2.2 THREATS

The dependability of cloud computing is put at risk by various threats. These threats correspond to the various types of faults that are transversal to every computer system: software, operator and hardware faults. Hardware faults are a daily occurrence in cloud computing deploy-

ments [128], being no less common than what has been reported for other large deployments, such as HPC [122]. Nevertheless, they represent a minority of all the fault occurrences in cloud computing. In fact, evidence suggests that operator and software faults dominate the pool of occurrences of faults [20, 52, 60, 61, 87], particularly when considering only public cloud deployments instead of private clouds [23].

This observation can be explained by a characteristic of cloud computing: the significant increase in software that is used to provide consolidation through virtualization and to manage the complex distributed systems that are cloud computing deployments. At the same time, the elastic and self-service nature of cloud computing, which calls for requesting and releasing resources by the client and without human interaction, the heterogeneity of the workloads that are executed, and the overall complexity of cloud computing deployments lends to the increase in operator faults, which often are caused by cloud clients that fail to interact with the user-exposed interfaces, instead of operators working for the cloud provider that perform incorrect operations.

The aforementioned threats eventually lead to failures, many of which cause visible outages of the provided service. Between 2009 and 2016, outages in the cloud are estimated to have occurred an average of three times or more per year in almost half of all big public cloud providers. However, in the worst years the number of outages can be as high as eight per year [60]. These outages translate to downtime that, in turn, cause cloud providers to violate their SLAs, namely regarding the clauses that specify the provided level of availability. In fact, the promising goal of five nines (99.999%) availability in the cloud has yet to be accomplished, since most cloud providers fall short of even 99.9% availability, with a small minority of providers not even reaching 99% availability in an average year [60].

The trace log dataset of Google Cloud nodes [115, 154] is a useful data source that provides information about failure rates and related metrics in a public cloud provider. Using this dataset, it has been inferred that the studied cluster, which features ~12500 individual server nodes, experienced an average of 309 server failures per day, with a standard deviation of 101 failures, which results in a single-node MTBF ranging between 12 and 13 hours [52]. Another relevant observation is that server MTTR ranges between 1 and 9 hours [52], which suggests that recovery actions are mostly manual, either because the failure can only be corrected with human intervention (*e.g.*, a failed hardware part had to be replaced) or because there is a lack of automatic failure detection and recovery mechanisms put in place.

2.2.1 *Software Faults*

Software faults are often considered to account for the majority of faults that affect cloud computing. Different studies attribute a wide range of failure percentages caused by software faults, ranging between 87% [61], 40% [87], 31% [20] and 15% [60]. This variance may be explained by the different methodologies used in the papers, the source of information and the fault types that were considered in their analysis.

Software faults are inevitable in software applications, as software testing and verification approaches cannot guarantee the complete absence of faults. Moreover, software faults become more common as the number of lines of code increase. Given the widespread use of new software that is designed to aid cloud computing in its endeavors, such as to provide virtualization or manage cloud deployments, the logical outcome is an inevitable increase of the number of software faults and a percentage gain relative to hardware and operator faults.

Cloud computing involves letting the clients manage resources and operations without direct human-interaction, usually through an interface. Software faults in these interfaces, which are called cloud APIs, have been the subject of extensive analysis due to their importance and recurrent usage to satisfy most of the requirements related with managing the deployments of the cloud client [85, 89, 96, 158].

Cloud management software is another software component essential to cloud computing that is not usually found in other large deployments. This type of software is used by medium to large cloud providers to manage their infrastructure. Software faults in this component tend to be activated only using specific and somewhat intricate combinations of inputs and events, hence having a low likelihood of being detected during software testing, but can cause failures with a big impact in the system, including causing single and multiple node failures (about 64% of failures) and catastrophic cluster-wide failures (about 36% of failures) [158]. Not only can software faults in cloud management affect the entire cloud infrastructure at once, but they also tend to cause failure modes that exhibit incorrect output being produced (about 66% of failures), service crashes (about 14% of failures) or state inconsistency (about 5% of failures) [158].

Studies have also shown that errors can propagate undetected inside the cloud management software and its various components before leading to a failure, and that a large percentage of these failures are wholly undetected or, if detected, are not reported to the user, thus increasing the chance of silent data corruption and inconsistency in the state [37]. Given the complex nature attributed to bugs in cloud management software, their failures fit perfectly under the definition of emergent failures [53], which are those failures that are born due to various fault activations and evolve into unexpected failures that had

not been planned for during design time due to the high complexity and high degree of interaction between components that is required for their occurrence.

In general, the failure modes of software faults vary with the faulty software component but tend to result in simple failed operations (42%), performance problems (23%), crashes or hangs that lead to unavailability (18%), data loss (7%) or data corruption (5%) and data staleness (5%) [61]. Similarly to what has been reported for software faults in cloud management software, software faults in other components that are part of the cloud infrastructure can lead to failures that affect and disrupt the entire cloud service [61]. Failures in cloud APIs tend to cause either halt failures (35%) or failed calls with error message (about 27%), but can also cause timing failures where the answer arrives after the specified time interval (about 6%) and missing or wrong content being returned (about 3%) [85]. Another interesting observation found throughout various papers in the area is that several documented outages have occurred due to failures being undetected, thus recovery action was never automatically triggered, failure detection incorrectly classifying events as failures (*i.e.*, false positives) and prematurely triggering recovery action [60, 87], or recovery action that fails [60]. This fact supports the importance of error and failure detection mechanisms [60, 61, 67] and verifying them during development and production phases at a cloud scale [61], such as using fault injection to emulate errors or failures that should be detected by these mechanisms.

2.2.2 *Operator Faults*

Historically, operator faults have been a common cause of failures in different domains such as HPC deployments, clusters, Internet services and other complex distributed systems [97, 101, 105]. That observation remains true when applied to cloud computing [62] as, in fact, operator faults have become more common in cloud computing due to many of the micro-management operations (*i.e.*, management of the resources of a client, in opposition to the management of the cloud computing infrastructure) having passed from qualified operators to possibly unexperienced cloud clients that have to interact with cloud APIs to perform the action that they desire.

Several works have classified operator faults into process errors and configuration errors [101]. Process errors include incorrect actions, actions that should have been performed but were not or actions that were taken in an incorrect order. Configuration errors refer to errors in the formatting of the configuration and errors in the used configuration values.

In cloud APIs, an estimated 30% of failures are due to operator faults, namely configuration errors that are often due to misconfigured

parameters [89]. When discussing operator errors in cloud APIs, the operator refers to the cloud client and not to an operator belonging to the cloud provider. In terms of impact, operator faults in cloud APIs tend to cause mostly content failures (55%) and halt failures (35%), which result in the desired operation being unsuccessful, in the case of content failures, and hanged operations that never complete, in the case of halt failures [89]. Indirectly, the above failures can lead to downtime of the provided service.

Research about how operator faults can affect the dependability of cloud computing deployments is limited, despite their importance. However public cloud providers occasionally release reports detailing the root causes behind large outages that have affected these providers. In 2011, Amazon EC2 suffered an outage in the provided service of any VM that used a set of affected storage resources [142]. The root cause behind this outage has been identified as an incorrectly performed change in the network configuration with the objective of redirecting traffic to another network. In 2012, Amazon ELB suffered an outage due to one of its developers having inadvertently deleted the state data during a maintenance process that was wrongly performed against the production server [143]. In 2017, Amazon S3 suffered disruption due to an operator mistake that caused a larger than planned number of servers to be removed from a subsystem, which lead to an overload of that subsystem and subsequent service failure [144]. In summary, the available information suggests that operator faults may cause widespread service outages, which tend to have a high repair time, and can also lead to data deletion by mistake.

2.2.3 *Hardware Faults*

Hardware faults happen every day at large cloud datacenters. Although the assumption that is often employed is that hardware only (or almost always) experiences permanent full-stop failures, reality has shown that hardware has a range of failure modes other than no failure or full-stop failure [67]. Hardware may limp (*i.e.*, operate in a degraded state) or suffer errors and failures that are not reported to the user [61]. These type of failures often arise due to transient or intermittent hardware faults, in opposition to the permanent hardware faults that tend to generate full-stop failures that imply replacing the faulty hardware component.

Among all the hardware that usually exists in a large datacenter, hard drive disks (HDDs), DRAM and power supplies (PSU) tend to be those that account for the highest number of permanent failures. Some reports indicate a replacement or failure percentage (depending on how the data was collected) of 20-78% for hard disks, 3-30% for DRAM memory and 9-12% for PSUs [23, 122, 148]. CPUs are reported to not fail often, with replacement percentages near 2%, however it

is well documented that CPUs are susceptible to transient hardware faults due to radiation, electromagnetic interference and temperature stress, which are hard to detect and do not cause permanent failures (hence the low reported numbers). Nevertheless, transient faults can have devastating effects on the availability and reliability of the system, including causing silent data corruption.

Many of these transient hardware faults, whose outcome is often called as "soft error" due to its temporary nature [94] (*i.e.*, the error disappears when a new value is written over it), occur when an ionizing particle strikes a sensitive element of a semiconductor device, which can cause a charge disturbance strong enough to flip the value that was stored in a flip-flop, latch, register, or memory cell [17]. In cloud applications, the most common causes of soft errors are alpha particles resulting from impurities in packaging materials and cosmic rays that enter the Earth's atmosphere from the outer space [17, 95].

Soft errors scale up with the amount of transistors and, indirectly, with the amount of hardware components in the datacenter, which will continue to increase in the future [77, 127]. Furthermore, soft errors will become more prevalent as technological developments lead to reduced node sizes in successive generations [27, 40, 94, 99]. Hence, the probability of occurrence of a soft error (*i.e.*, the soft error rate) is expected to continue increasing in the future [69].

DRAM and SRAM memory can also be affected by soft errors [135]. In fact, server-grade DRAM memory is already equipped with error correcting codes that detect and correct the large majority of soft errors. Nonetheless, soft errors can still defeat these mechanisms (some reports state a FIT value for undetected errors in server-grade DRAM of up to 21.7 per device, with a mean value nearing 1 [135]) and cause multi-bit errors. SRAM is even more prone to soft errors than DRAM, particularly to soft errors that cause single bit-flips, and the most affected SRAM components of a system tend to be also the larger ones, such as L2 and L3 caches and the translation lookaside buffer (TLB) [135], although soft errors hitting the smaller CPU registers has also been documented.

Another challenge to the dependability of hardware in a datacenter that is going to become more common in the future is the reliability of SSDs. SSDs are already replacing HDDs in datacenters due to their high-performance, low power usage and ever lower acquisition costs [4]. However, SSDs have been shown to be less reliable than HDDs (*e.g.*, a certain datacenter saw almost twice as many SSDs replacements than HDD [108]), particularly with regards to uncorrectable errors [123]. Uncorrectable errors represent one of the most dangerous type of errors in storage devices because they lead to data loss or corruption.

While HDDs are said to experience an annualized failure rate (AFR) between 1.7% and 8.6%, along with a 2% probability of a HDD experi-

encing CRC errors during its lifetime [109, 125], SSDs have a slightly worse AFR that ranges between 4% to 10% [123]. Furthermore, between 4 to 24% of all SSDs experience at least one uncorrectable error during their lifetime and almost every SSD has at least one bad block. The raw bit error rate of SSDs in production is said to range between $5.8 * 10^{-10}$ and $3 * 10^{-8}$ [123], whereas specifications indicate raw bit error rate of HDDs around 10^{-13} to 10^{-16} [57, 74]. Although failures that cause silent data corruption in HDDs are less likely than in SSDs, they still occur [10, 45, 111], despite most fault tolerance mechanisms not providing coverage against such failures [102], due to failures of the HDD itself and interferences during data transfer (*e.g.*, exposed cables), hardware faults in memory and cache that affect the transmitted data, or software faults in the firmware of the drives or RAID controllers [70, 74].

In summary, hardware faults in large datacenters, like those that support cloud computing, are commonplace and tend to cause fail-stop failures (*e.g.*, a HDD stops working) that force the replacement of the affected component and require redundancy to tolerate its outcome (*e.g.*, RAID arrays to avoid data loss if a disk stops working). However, less-known soft errors and other transient and intermittent faults can also affect the hardware, causing possible data loss or silent data corruption. The fact that some hardware faults may cause silent data corruption turns them into a threat that is not matched by either software and operator faults, since these two tend to cause unavailability rather than data corruption. This means that hardware faults deserve special care when designing fault tolerance mechanisms, which, in our opinion, has not been properly addressed in cloud infrastructures.

2.3 MEANS OF ATTAINING DEPENDABILITY

Dependability can be attained by employing fault prevention, fault removal and fault tolerance. Fault prevention acts the earliest in the development cycle, in order to detect and reduce the amount of faults that are introduced in the system. Fault removal happens during development, using testing techniques to detect and remove faults that have been introduced, or during use, by employing corrective measures. Since programmers that develop the applications that run in the cloud rarely focus on non-functional requirements, it is less likely that they employ effective fault prevention and removal approaches, thus dependability has to be provided by the cloud provider or cloud client through fault tolerance techniques. Furthermore, the employed fault tolerance techniques depend on adequate error detection and recovery mechanisms that should be adapted to the characteristics of cloud computing, namely by taking advantage of virtualization and the distributed nature of the cloud.

Error detection can be a daunting task due to the complexity of cloud deployments and the fact that the used workloads are very heterogeneous and managed by the cloud clients, thus being outside of the purview of the cloud provider. As such, it is common for failures to occur due to incorrect or insufficient error detection [60, 61, 67, 87]. After an error is detected, the cloud provider can attempt to recover the affected component before the error evolves into a failure. This can be the case when a node starts to show unstable behaviour (*e.g.*, a DIMM memory fails into a limp state and its error rate increases substantially) and the provider decides to decommission the node for repair (which can imply a manual root cause analysis and repair process) by first migrating all the VMs that were hosted in the affected node over to the other nodes in the cluster. This is an example that shows how the distributed and highly-scalable nature of cloud computing deployments, as well as the containerization and migration capabilities of virtualization, can be used in favor of ensuring dependability.

After all VMs have been migrated out of the node, cloud providers will often attempt to reboot it before performing a manual analysis and repair in an effort to fix transient errors, such as those caused by memory leaks and other software faults. This process of "gracefully terminating an application and immediately restarting it at a clean internal state" [68] is known as software rejuvenation and can equally be applied at node-level or at a higher level (*e.g.*, VM-level, application-level). Its effectiveness derives from clearing memory leaks, resource leaks and other state inconsistencies that long-running processes and high-uptime systems develop over time and which lead to performance loss and service failures if left unchecked. Microrebooting is a related technique that performs "individual rebooting of fine-grain application components" [22] to take advantage of the error recovering properties of rebooting while reducing the downtime associated with the reboot process.

Most fault tolerance mechanisms depend on support from the cloud provider, but cloud clients can, in certain cases, configure advanced mechanisms that are implemented by the cloud providers but are not enabled by default. An example is availability zones, which consist of distinct and isolated datacenters spawned across different geographical locations and which can be assigned, by the client, to specific VMs or other resources hosted in the cloud. The advantage behind setting up multiple availability zones is that, if one of the zones fails, the VMs that are running in the remaining zones can continue to operate correctly and may take the load of the failed instances, assuming that the client has configured the failover mechanisms to do so.

More advanced but less common approaches are available to private and community loud providers that undertake the effort of installing and configuring them, although their usage in larger public providers

is rare at best. One such approach relies on VM checkpointing between an active and a passive node that are executing in different physical machines through the exchange of VM state over the network. Whenever the active node suffers a failure (*e.g.*, a hardware failure) that disrupts the provided service in a way that it stops responding to periodic heartbeats, the passive node detects and takes its place from the last known state. The best known example of such an implementation is Remus [38]. However the usage of a technique like Remus has performance and cost disadvantages which need to be considered, such as requiring twice as much hardware and respective energy consumption (although the passive node sees lower usage than the active node), increased network traffic due to state synchronization, and performance overhead related with synchronization.

A similar technique that relies on synchronizing and duplicating VMs across different physical machines is COLO [41], which uses a setup where both nodes are active and the inputs are replicated across them, although only the primary node interfaces with the outside directly. COLO constantly monitors the output of both VMs and temporarily suspends their operation to synchronize the secondary VM with the state of the primary VM whenever a discrepancy is found, in order to ensure that both VMs share a functionally equivalent state, which may nevertheless differ due to non-determinism as long as it does not lead to different output being produced. It suffers from the same drawbacks as Remus and even experiences higher hardware utilization and energy consumption due to its 'two active nodes' setup, but promises better performance and scalability than Remus due to less strict synchronization and replication mechanisms. This technique covers only fail-stop failures caused by hardware faults, which are incorrectly assumed to be the only possible failure mode, and disregards other possible faults.

Existing lightweight techniques to protect virtualized and cloud computing deployments tend to employ microrebooting both in a reactive and proactive manner to recover from state corruption and other latent errors that may lead to failures. ReHype [83] performs microrebooting of the hypervisor by temporarily pausing the execution of the VMs while the underlying hypervisor is being restarted and then performing a re-connecting between the rebooted hypervisor and the previously running VMs. It has been shown to be able to recover at least part of the VMs of the system after a failure caused by a soft error. HyperFresh [9] performs software rejuvenation of the hypervisor state by remapping the VMs from a old to a new hypervisor instance in a matter of milliseconds.

Manual intervention will be required if everything else fails and the failure cannot be recovered using automatic measures, which incurs a significant MTTR and lowers availability. This can be the case when hardware needs to be replaced, when a operator fault leads to changes

that must be manually reverted, or when a software fault has to be patched out.

Most of the aforementioned fault tolerance techniques cover failures caused by permanent hardware faults that cause outages and fail-stop crashes of the service. With the exception of the techniques that perform microrebooting, which cover a subset of transient hardware faults and software faults, the available techniques are wholly inadequate to tolerate the failures caused by transient hardware faults, software and operator faults, which are precisely the types of faults that have increased in likelihood due to cloud computing. For this reason, there is a clear need for new fault tolerance techniques that can be applied to cloud computing, preferably taking advantage of the unique characteristics of cloud infrastructures, as proposed in the present thesis.

2.4 SUMMARY

Cloud computing is unique because it possesses several characteristics that are not shared with other computing paradigms. The main identifying characteristics of cloud computing that have an impact in its dependability are:

- Consolidation of tenants over the same physical hardware;
- Usage of virtualization to support tenant consolidation;
- Resources are acquired and released by the cloud client and without human intervention through automated APIs;
- Heterogeneous workloads that are not controllable by the cloud provider;
- Size of cloud computing datacenters;
- Complexity of cloud computing deployments;
- Usage of energy saving and cost reduction techniques;
- Primary focus is availability, reliability comes after.

The threats that affect the dependability of cloud computing are hardware, software and operator faults. Software and operator faults dominate the pie of occurrences and their high probability is specific to cloud computing, arising from its increased complexity and higher dependence on software. Hardware faults represent a minority part of all occurrences but are nonetheless frequent and will continue to increase with the expansion in size of datacenters and advancements in microprocessor manufacturing.

The outcomes of these faults can consist in outages that cause unavailability in a single or multiple nodes, cloud-wide unavailability,

data loss, data corruption, among other failure modes. Table 1 provides an overview of the threats that affect the cloud and the failure modes that each threat can cause.

THREATS	REASONS	FAILURE MODES
Software faults	Cloud API bugs	Failed Operations
	Memory leaks	Performance Problems
	Latent faults in hypervisor	Downtime
	Latent faults in CMS	Data Loss
	Other software faults	Data Corruption
Operator faults	Configuration faults	Downtime
	Process faults	Data Loss
Hardware faults	Permanent faults (DRAM, PSU, ...)	Downtime
	Transient faults (Soft errors)	Data Loss
	Intermittent faults	Data Corruption

Table 1: Summary of threats to the dependability of cloud computing and failure modes that they cause.

Many outages can be attributed to errors in components that remain undetected during large periods of time and eventually lead to gray failures. This kind of failures exposes a limitation common in cloud computing deployments, the fact that the state of the system is assumed to be either correct or failed, without taking into account that there are other possible states in-between in which the system appears to operate normally but is in reality working in a degraded mode.

In general, to defend the dependability of cloud computing against the aforementioned threats, error detection and fault tolerance mechanisms must be employed. The currently available and commonly employed fault tolerance techniques for cloud computing are lacking in the coverage of faults other than permanent hardware faults. This fact is specifically preoccupying given that software and operator faults, and to some extent, transient hardware faults, are expected to increase the most going forward and already represent the large majority of faults that affect cloud computing. Thus, new fault tolerance techniques must focus in providing coverage against a broader spectrum of faults, all the while minimizing performance overhead and energy consumption.

Part II

EVALUATING CLOUD COMPUTING DEPENDABILITY

3 EVALUATING DEPENDABILITY OF CLOUD COMPUTING

The evaluation of dependability and its associated attributes is a key step in the design and evaluation of reliable software systems and fault tolerance mechanisms. Availability, despite being only one of the attributes that form dependability, is the attribute most often used in cloud computing to measure the quality of the provided service [16]. Cloud computing providers publish service level agreements (SLAs) which specify, among other aspects, the availability (or uptime) expectable from their platform and below which the cloud provider becomes liable to compensate the client. As an example, at the time of writing the SLA used by the majority of Amazon EC2's customers indicates that clients are entitled to compensation whenever the provided monthly availability is lower than 99.99% (*i.e.*, about 4 minutes of downtime each month), with increasing levels of compensation up to 95% availability (*i.e.*, 36 hours of downtime in a month) [2].

Evaluating dependability is not straightforward, because failures represent the rare exceptions to the norm, rather than the normal behaviour of the system. They represent statistical outliers and the acquisition of sufficient failure data to estimate coverages, such as fault handling and error isolation coverage, to identify failure modes and, in general, to verify hypotheses with sufficient statistical confidence is difficult to perform with failure data from production systems. Therefore various approaches have been developed to perform and accelerate dependability evaluation, such as fault injection [5], which is the principal technique used in this thesis.

Fault injection consists on the insertion or emulation of faults in a system or a model thereof, with the objective of producing failures that are representative of real-world failures, but at an accelerated rate (*e.g.*, fault injection can produce failure data in a few days comparable to years of production failures). Although fault injection is an established technique, with decades of research and usage in academy and industry alike, only a limited number of works have focused on the intersection between cloud computing and fault injection. Moreover, most available fault injection tools are not adapted to evaluate cloud computing systems, namely due to the lack of support for orchestrating campaigns across multiple nodes and to inject faults in a virtualized architecture.

In this chapter, we present the ucXception framework, which combines a suite of fault injection tools capable of injecting various fault models with pre-made code for automating and managing fault in-

The ucXception framework is an open-source project which is available at `https://github.com/ucx-code/ucXception`

jection campaigns, probing and extracting information from local or distributed nodes and analyzing results. This framework and, specifically, its fault injection tools supported the experimental evaluations described in Chapters 4, 5 and 6. When comparing the ucXception framework with other available fault injection tools and frameworks, three main differences exist:

1. ucXception enables the injection of hardware faults and software faults under the same framework;
2. ucXception natively and transparently supports execution across multiple nodes of a distributed system;
3. ucXception has native support for injecting faults in virtualized systems, including in the hypervisor and VMs.

This chapter starts with an explanation of concepts related to fault injection and proceeds to introduce the ucXception framework, with particular focus to how it is tailored for evaluating cloud computing and virtualized systems and the various fault models that it supports.

3.1 BACKGROUND IN FAULT INJECTION

Fault injection can be accomplished using different approaches. Some approaches depend on external hardware while others are implemented partially or entirely in software. Hardware-implemented fault injection (HWIFI) acts directly on the hardware to perform fault injection, whereas software-implemented fault injection (SWIFI) is a more recent variation that employs software approaches, such as software traps, running the target application in trace mode, or modifying the binary executable prior to execution, to perform fault injection in a generic way across different hardware architectures. Fault injection can also be performed in models or simulations of the target system, which can be particularly useful when the real system is not yet developed or otherwise unavailable. A mix of the aforementioned approaches can also be used, which is called hybrid fault injection.

In general, the approach used to perform fault injection can be classified according to the following properties [12]:

- *Controllability* – the ability to control the injection in time and space;
- *Observability* – the ability to observe the effects of the injected fault;
- *Repeatability* – the ability to repeat the fault injection experiment and obtain the same result;
- *Reproducibility* – the ability to reproduce the results of a campaign;

- *Faultload representativeness* – how accurately the fault represents real faults;
- *Workload representativeness* – how accurately the workload represents real system usage;
- *System representativeness* – how accurately the target system represents the real system.

It becomes a theoretically impossible problem to obtain a fault injection approach that excels at every single property, thus fault injection approaches balance trade-offs depending on their desired application. Usually, SWIFI affords higher controllability, repeatability and reproducibility than HWIFI, but model and simulation-based fault injection should perform the best in these properties. On the other hand, HWIFI is less intrusive than other approaches, thus resulting in better system representativeness than SWIFI and, of course, model and simulation-based approaches, where the evaluated system is an approximation of the real system.

Fault injection is usually applied with one or more of the following goals: i) identify failure modes and estimate their probability of occurrence; ii) estimate fault coverage or other coverages and validate the effectiveness of fault tolerance mechanisms; iii) analyze fault propagation. Measuring failure modes and their probability of occurrence is one of the principal goals for which fault injection was employed in this thesis and, in general, enables a characterization of the system and how it fails, which is essential for designing fault tolerance mechanisms that increase the dependability of said system.

Fault coverage, or fault handling coverage, is one of the various metrics that can be evaluated using fault injection and which measure the effectiveness of fault tolerance mechanisms [7]. For example, in this thesis fault injection is used to estimate error isolation coverage of the Xen hypervisor (in Chapter 4), which reflects how well the hypervisor is capable of preventing an error in one VM from propagating to the remaining components of the system, including other neighboring VMs.

Fault propagation analysis strives to track the path that a fault follows inside the system since its activation until it causes a failure. Depending on the system being studied an error may cross the boundaries of various components (*e.g.*, an error in the hypervisor may propagate to a VM through the return code of a hypercall), hence understanding the mechanisms and the components behind this occurrence enables the design of better isolation mechanisms.

3.2 THE UCXCEPTION FAULT INJECTION FRAMEWORK

The ucXception framework was developed to be used in the evaluation of cloud computing and the various components and layers that form

a virtualized system. For this purpose it includes a fault injection tool capable of emulating transient hardware faults in a VM or a hypervisor and was designed to transparently execute its steps in a distributed system as if it was executing them in a local system. For example, a user can configure a fault injection campaign that targets an application running on the same computer as ucXception, like shown in Figure 4. Whereas just adding a new entry to a specific data structure (remote_hosts) and changing part of one code instruction, as shown in Figure 5, is enough to reproduce the same campaign but targeting an application in a remote node.

```
1   fi = (local_hw_fi, "localhost", "<injector_path>")
```

Figure 4: Example of configuring fault injection locally.

```
1   remote_hosts = {
2     "node1" : ssh("192.168.1.5", "root"),
3   }
4
5   fi = (local_hw_fi, "node1", "<injector_path>")
```

Figure 5: Example of configuring fault injection in a remote node.

One of the guiding principles of ucXception was allowing the non-expert users to design and tailor a fault injection campaign with a reduced amount of modifications and configurations. To accomplish this objective, the framework includes multiple out-of-the-box elements that implement specific functionalities and which can be mixed and matched according to requirements, as well as including a base campaign template that can be adapted and various examples of campaign configurations. In fact, it should be possible for most campaigns to be created solely by performing modifications to the existing variables in two separate source files, thus freeing the user from having to delve in the remaining source files.

Figure 6 presents a high-level overview of the architecture of ucXception, where we group the elements into elements that the user should change to configure a new campaign, core components that will be reused and the outputs from a campaign.

3.2.1 *Components of the ucXception framework*

The ucXception framework revolves around a set of pre-made components that provide contained functionality and which can be connected together to solve the user's needs. These components can be described as:

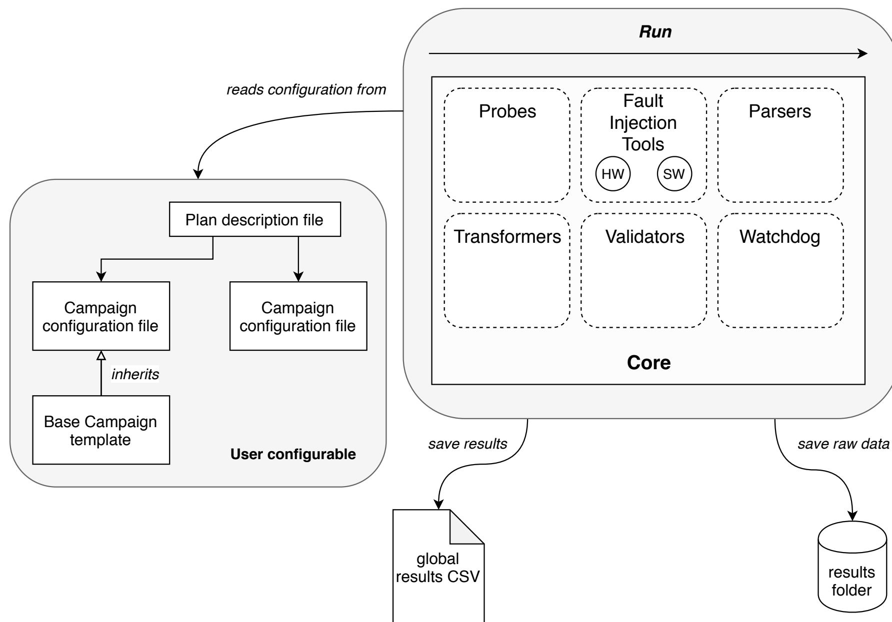


Figure 6: Architecture of the ucXception framework.

- `Plan` – A plan consists in a set of campaigns. It represents a high-level experiment that has been designed by the user;
- `Campaign` – A campaign consists in a set of runs of a specific campaign configuration (a Python file with logic and configuration options). Different runs can have different parameters, *e.g.*, some runs may perform injection while others will not (golden runs);
- `Run` – A run represents a single execution of the experiment flow defined in the campaign configuration, using the given run-specific parameters and campaign-specific values;
- `Watchdog` – A watchdog should be used to monitor the execution time of a run to ensure that it does not extend over the user-defined alloted time. If the run is taking too long and since it may even never end (*e.g.*, the application has entered an infinite loop), the watchdog should kill the workload application and note the occurrence;
- `Probe` – A probe represents an application that will be launched for the duration of the run that has the purpose of monitoring and storing information relative to the system or application being evaluated. Probes can be sub-divided into pre- and post-probes, according to whether they are launched before or after the workload has started. Pre-probes usually collect system-wide metrics, whereas post-probes monitor specific processes, hence the need for post-probes to be launched after the workload;

- `Fault injection tool` – The fault injection tool implements a specific fault model (*e.g.*, single bit-flip for emulating soft errors) and allows the emulation of faults according to that model;
- `Validators` – The validators are small pieces of code, usually Python functions, that inspect the results obtained during a run and verify acceptance conditions. If a validator fails (*e.g.*, fault injection was not successful) then the data for that run will not be written to disk;
- `Parsers` – A parser is a function that reads the results of the run (*e.g.*, output from the fault injection tool) and converts them into a more useful and compact format. The output from the various parsers is stored in the results CSV file;
- `Transformers` – Transformers are similar to parsers, in the sense that both receive raw input and produce a condensed output, but whereas the output from the parsers is stored in the results CSV file, the output from transformers is stored as individual files in the run's own result's folder. Transformers are mostly used to convert the raw output of the probes into a more manageable format, *e.g.*, converting a binary data file originating from a resource monitoring probe into a CSV file where each row represents a time interval and each column represents a monitored resource.

The ucXception framework natively possesses various components, however users are free to create their own components to fulfill an unanswered need. In terms of pre-probe components, the following are available out-of-the-box:

- *Logs probe* – A simple probe that extracts logs from the target system during the *Post finish* phase. It is ready to extract logs specific to Linux, Xen and Openstack. The user can easily configure it to support other types of logs;
- *IntelPCM probe* – Intel PCM (Processor Counter Monitor) [34] provides a way of monitoring hardware counters in recent Intel hardware. This probe can be used to monitor the CPU, memory and power counters throughout the run;
- *Ping probe* – A simple probe that performs pings at a user-specified interval between a source and a target computer and stores the results, along with the timestamps. Can be used as a rudimentary way of monitoring the state of various systems;
- *SAR probe* – SAR [55] is an utility that uses the various interfaces provided by the Linux kernel in order to monitor system-wide activity information, such as CPU, memory, network, disk or power metrics. The probe takes a snapshot of all the available

metrics using an 1 second interval (the lowest possible) and stores the results in a binary file;

- *TCPDump probe* – Monitors and stores all the network traffic in a specific interface. Supports passing TCPDump [141] rules to filter the packets that are captured;
- *Xentrace probe* – Xentrace [46] is an utility that monitors the events that occur in a Xen virtualized system. The results are stored in (usually large) binary files.

With regards to post-probes, only one such component is currently available:

- *Pidstat probe* – Somewhat similar to the *SAR probe*, since it also captures similar metrics, but focuses on a specific process (whereas SAR is system-wide). Can be used to monitor the workload application.

In terms of parsers, the following are available:

- *HW FI parser* – Reads the output produced by ucXception's Linux-based HW fault injection tool, which emulates transient hardware faults, and stores the register, the bit, the injection time, the PID of the process that was affected by the injection and the pre- and post- injection values of every register;
- *SW FI parser* – Stores information relative to the injection performed by ucXception's SW fault injection tool, such as the applied operator or in which line the fault was injected;
- *Pcap -> TCP parser* – Reads the data from a *TCPDump probe* and very quickly calculates its statistics, such as total packets, total packets by type (RST, FIN, ...), retries, and others;
- Info parser - Stores generic information about the run, namely, the run number, its start and end time, and duration;
- *MD5 output parser* – Obtains the output of the workload application and computes its MD5 hash. Compares the obtained MD5 hash against a fixed, expected hash and stores whether both hashes match and the size of the produced application's output. Useful to detect silent data corruptions whenever the workload application produces a deterministic output (*i.e.*, always produces the same output when it receives the same inputs);
- *Return code parser* – Stores the return code of the workload process. Can signal a successful termination or an abrupt termination (*e.g.*, killed by the operating system due to a segmentation fault);

- *Current folder parser* – Minimalistic parser that just stores the path of the results folder of the current run.

In terms of transformers, the following are available:

- *Pcap -> TCP 2 CSV transformer* – Uses the Tshark application to convert a PCAP dump of network traffic into a CSV file with high-level information about each packet, such as the TCP flags, packet size, IPs and ports, or timestamps;
- *Pidstat 2 CSV transformer* – Converts the binary file generated by the *Pidstat probe* into a CSV file;
- *SAR 2 CSV transformer* – Employs the sadf utility [55] to convert the binary file produced by *SAR probe* into a CSV file;
- *Ping 2 CSV transformer* – Converts the output of the *Ping probe* into a more structured CSV file;
- *Save output transformer* – Saves the raw output (stdout and stderr) from the workload application into files. Can be used when a more detailed analysis to this output is required, or for debugging.

There is one available validator, called *Ensure Injection*, which checks whether one and only one injection (of the ucXception's Linux-based HW fault injection tool) has occurred in a run by comparing the pre- and post-injection values of all registers and ensuring only one bit of one register has changed.

3.2.2 *Fault injection tools of the ucXception framework*

ucXception comes equipped with three fault injection tools that implement different fault models. There are two different fault injection tools for emulating hardware faults, which operate in a different manner and target different systems, and one fault injection tool that emulates software faults. Table 2 summarizes and indicates which tools were used to perform fault injection campaigns in the various chapters of this thesis. Once again, if the user desires he can easily integrate other fault injection tools into the framework.

3.2.2.1 *Hardware fault injection in Linux-based systems*

This fault injection tool emulates soft errors that affect the CPU's register file or other components of the CPU (buses, ALU, FPU, *etc.*) by implementing the single bit-flip fault model [71, 116]. Bit-flips are restricted solely to CPU registers and there is no support for directly performing bit-flips in the memory. Injection in memory words is not implemented due to the existence and popularity of very effective

DESCRIBED IN	EMPLOYED IN	TARGET	FAULT MODEL
§ 3.2.2.1	Ch. 4	Linux Applications and Kernel	Soft Errors
§ 3.2.2.2	Ch. 6	Any VM (including hypervisors)	Soft Errors
§ 3.2.2.3	Ch. 5 & 6	Applications written in C and C++ language	Software Faults

Table 2: Fault injection tools of ucXception.

ECC for memory [124] and because part of the soft errors affecting the memory can be accurately emulated using fault injection in CPU register files.

The tool can run in any modern Linux kernel and supports the x86_64 and ARM architecture. When used in a x86_64 system it can inject on the *rip*, *rsp*, *rbp*, *rax*, *rbx*, *rcx*, *rdx*, *cs*, *ss*, *ds*, *es*, *fs*, *gs*, *eflags* and *r8* to *r15* registers, as well as, being able to inject in FPU and SEE registers. If executed in an ARM system it can inject on *sb*, *pc*, *lr*, *sp*, *ip*, *a1* to *a4* and *v1* to *v8* registers.

It employs the *ptrace* functionality available in practically every Linux installation and which is also the engine behind the famous *gdb* debugging tool, to attach itself to a running process, briefly suspend its execution, obtain the data structures of the Linux kernel that hold the process' register values, perform the bit-flip according to the passed parameters and resume execution. After the target process resumes execution, its register values will include the bit-flip. Since the tool is software-only and does not depend on any hardware extension or feature, we are referring to SWIFI (Software-Implement Fault Injection). Furthermore, since the injection can be performed without requiring any modification to the target program's source or binary code, it can be classified as a run-time approach.

The tool also includes logging functionality that stores the exact timestamp of the injection moment and the values of every register prior and post the bit-flip. This information is extremely useful not only to validate that injection is working correctly, but also to enable detailed and complex analyses of the results.

The moment of injection is always temporarily triggered, but there are two ways of defining the trigger: *timeout* and *deadline*. In *timeout* mode the user specifies how many milliseconds the tool should wait before it performs the bit-flip, whereas in *deadline* mode the user specifies a UNIX timestamp, including milliseconds, that defines the desired moment of injection, which the tool will attempt to obey as closely as possible.

The flow of this fault injection tool is depicted in Figure 7.

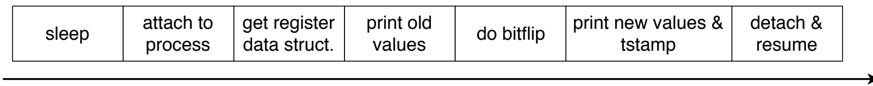


Figure 7: Flow of the fault injection tool for Linux-based systems.

The tool begins by sleeping the determined amount of milliseconds, accomplished through the *nanosleep* POSIX function. When the sleep time has been elapsed, the tool attaches itself to the process that the user wants to inject a fault in, extracts the register values for that specific process at that point in time, prints those values to the standard output stream along with the current timestamp, performs the bit-flip in the register and lets process execution continue.

3.2.2.2 *Hardware fault injection in virtualized systems*

A separate fault injection tool capable of emulating hardware faults was created specifically for use in virtualized systems. It is capable of injecting faults in any application running inside a VM, including a hypervisor as long as nested virtualization is used (*i.e.*, the hypervisor being targeted is executed inside a VM). The fault model remains the traditional single bit-flip in CPU registers and any of the *rip*, *rsp*, *rbp*, *rax*, *rbx*, *rcx*, *rdx* and *r8* to *r15* registers can be targeted.

The tool was implemented as a set of modifications to the Xen hypervisor that introduce a new hypercall and respective toolstack functions to control the fault injection process, as well as modifications to the scheduling subsystem to enable injections of faults inside VMs.

The injection process consists in modifying the register value stored in the data structure that holds a VM's CPU state and which is updated immediately prior to a context switch. This structure is needed because any hypervisor must know the latest state of the CPU between context switches of VMs. We take advantage of this fact to inject faults, but this means that the approach is dependent on the rate at which context switches occur, which is a configurable parameter in Xen. While higher context switching rates (*i.e.*, smaller timeslices) allow the fault injection tool to have a more precise moment of injection, they can also bring considerable performance overhead and intrusiveness to the system.

Furthermore, this tool is capable of filtering the application that is targeted for injection by looking at the value in the *rip* register (which points to the next instruction to be executed) and only performing injection whenever the *rip* is inside a user-defined range. This functionality can be specially useful if one wishes to perform fault injection that affects solely the hypervisor (or solely the non-hypervisor code) running in a VM, as there is a well established division between the virtual memory addresses assigned to the hypervisor, to the operating system and to the userspace applications.

Figure 8 presents the expected usage scenario for this tool.

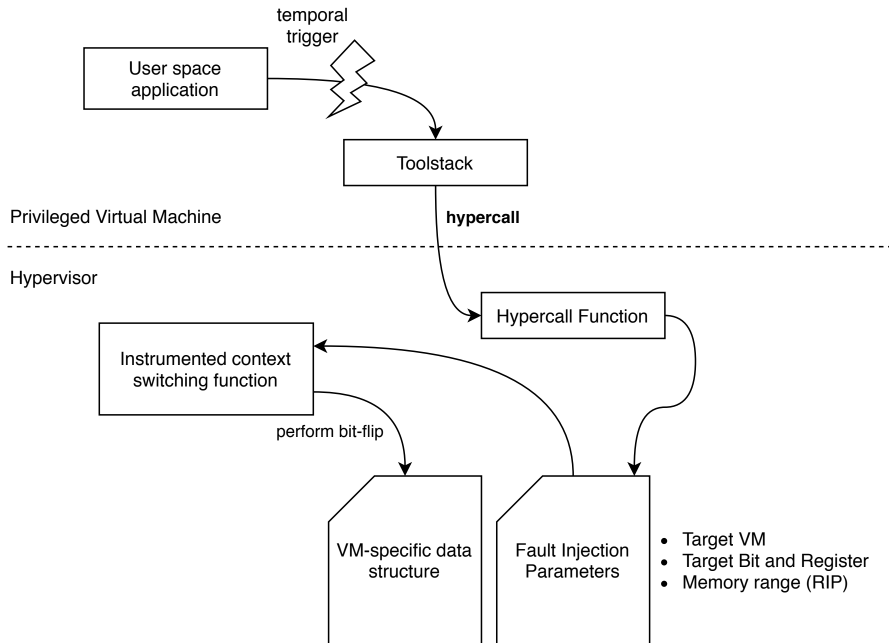


Figure 8: Flow of the fault injection tool for virtualized systems.

The flow starts from the PVM, where a user space application (or the ucXception framework) will provide the triggering functionality, which is not embedded in the fault injection tool, and will call the toolstack at the correct moment. The toolstack performs a hypercall to a function in the hypervisor, while passing the desired parameters for fault injection. These parameters include the target VM (when a system has multiple VMs, the tool can focus on just one of them), the target register and bit where injection will take place and the start and end of the memory range that the *rip* should be pointing at if injection is to take place, although this last parameter is optional. The hypercall function writes this information to an internal structure, which will be read during context switching, and if all conditions are met (the VM that is receiving CPU time is the same as the target VM and its *rip* is inside the expected range) the bit-flip is performed here, right before the target VM starts executing.

3.2.2.3 *Software faults in C source-code*

This tool performs software fault injection at the source-code level by applying program modifications (or mutations) that are representative of mistakes made by software developers [42], including some mistakes that tend to cause software vulnerabilities [11]. The faults defined by the fault model, whose operators are listed in Table 3, are injected in programs written in the C programming language and the injector itself is written in Java, using the Eclipse CDT plugin, which is the plugin that supports C/C++ programming in Eclipse. The fault injector takes as input the source code and CDT performs lexical and

syntactic analysis to produce the corresponding abstract syntax tree (AST). The software fault injector then traverses the AST to identify nodes in which faults can be injected, according to the operators and their constraints (*e.g.*, the statement to be modified is not the only statement in the code block). For each possible location/fault type pair, the tree is modified and the resulting program is then converted back into source code representation from which a patch file is generated and stored in the filesystem. This patch file represents the fault to be injected.

OPERATORS	DESCRIPTION
MFC	Missing function call
MIA	Missing `if` construct around statements
MIEB	Missing `if` construct plus statements plus `else` before statements
MIFS	Missing `if` construct an surrounded statements
MLAC	Missing `and` sub-expr. in logical expression used in branch condition
MLOC	Missing `or` sub-expr. in logical expression used in branch condition
MLPA	Missing localized part of the algorithm
MVAE	Missing variable assignment with an expression
MVAV	Missing variable assignment with a value
MVIV	Missing variable initialization with a value
WAEP	Wrong arithmetic expression in parameters of function call
WPFV	Wrong variable used in parameter of function call
WVAV	Wrong value assigned to a variable
WALR	Wrong algorithm – code was misplaced
WLEC	Wrong logical expression used as branch condition
EFC	Extraneous function call
EIFS	Extra `if` construct and surrounded statements

Table 3: Software fault model operators.

The default mode of operation produces patch files that contain solely the modified fault, however the tool also supports producing patches in an extended mode that supports enabling and disabling the activation of the fault (*e.g.*, the user might want to enable the fault activation only after certain steps have been carried out) and monitoring statistics about the fault, such as the timestamp when the fault was firstly activated and how many iterations occurred before and after the first activation. However, if this extended mode is used

certain operators, such as MVIV, cannot be emulated because they cannot be combined with conditional activation.

Figure 9 presents a typical flow when using this tool. The steps can be divided into two phases: the preparation and the execution phase. During the preparation phase the fault injection tool is called and provided with the application's source code as input, from which it will generate a large amount of patch files. During the execution phase, each of the generated patch files will be applied once at a time, the code with the patch applied will be compiled and the user-defined workload will be executed. After the workload has finished, results and data should be extracted and this logic starts again for the next patch file.

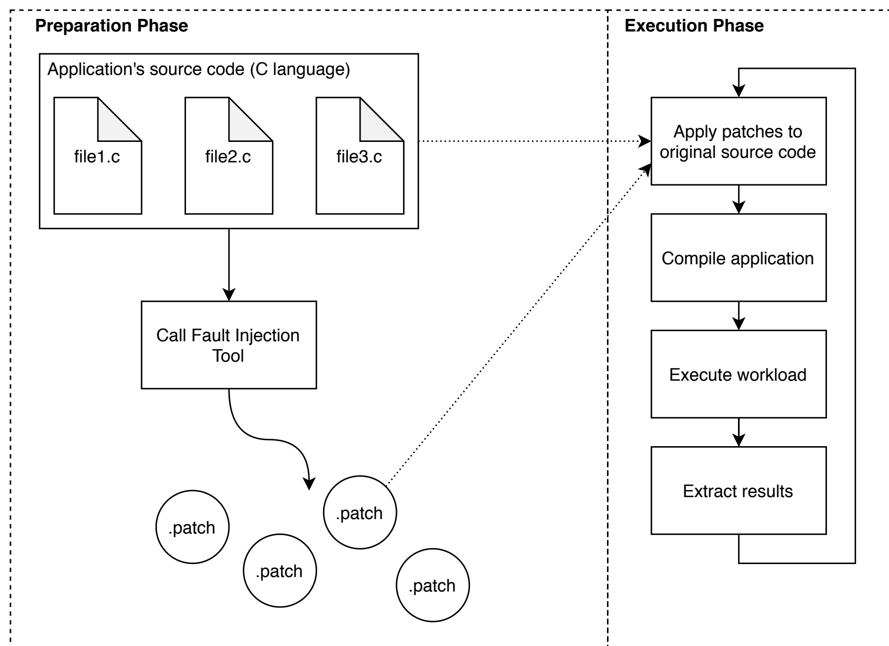


Figure 9: Flow of the fault injection tool of software faults.

An important aspect in software fault injection campaigns, specifically if their objective is to estimate failure modes and their probability of occurrence, is the representativeness of the injected faults. The majority of the produced faults does not reflect real faults that could be found in production systems, despite the usage of a representative fault model, because even basic software testing procedures, which are common in the large majority of software projects, would detect the fault before it reaches production. Residual software faults are those that escape the testing phase and thus reach production code and lead to failures.

Although neither the tool nor the framework include code specifically for filtering non-residual faults before injection, such can be obtained by including a pass where software tests are performed posterior to the fault being applied and compiled. Chapter 5 performed

this testing and filtering step and the results confirmed that a significant percentage of faults do not pass even basic testing, whereas Chapter 6 did not filter the injected faults because the main objective of the fault injection campaign conducted in that chapter was not to estimate failure modes or similar, but rather to evaluate a fault tolerance mechanism with regards to how it tolerates failures, caused both due to residual and non-residual faults.

3.3 RELATED WORK

Given the maturity of fault injection, several fault injection tools have been developed throughout the years. Thus ucXception does not represent a revolution in the area but rather an evolution over existing offers, namely due to the already referred aspects that make ucXception ideal for evaluating virtualized and cloud computing systems. Some of the most well-known fault injection tools are briefly described below, ordered according to their year of inception.

MESSALINE [6] was one of the first fault injection tools and used hardware-implemented fault injection at the pin-level, with configurable fault location, a temporal trigger and capable of injecting various transient and permanent hardware fault models, such as stuck bits.

FIAT [126] was developed for emulating errors generated by software and hardware and allowed the definition of when, where and for how long faults should be injected. It was capable of injecting faults in registers and memory, as well as communication faults in a distributed system.

FERRARI [72] is capable of emulating transient and permanent faults in software, such as bit-flips in memory and registers, and supports addition of other fault models. Faults may be injected according to a spatial trigger, using a software trap, or a temporal trigger.

FINE [73] focuses on measuring fault propagation rather than evaluating fault latency, fault coverage or failure modes. It is capable of emulating permanent and hardware faults, such as memory, CPU, bus and I/O faults, and software faults using a fault model that includes assignment, condition check, function and documentation faults. The tool takes advantage of the `ftrace` system call to inject the faults in a matter somewhat similar to the fault injection tool described in Section 3.2.2.1.

FTAPE [146] innovated by adding a synthetic workload generator and supporting stress-based injection, *i.e.*, performing one or more fault injection on the most stressed components or during time periods of higher system stress. It supports injection of single bit-flips in CPU registers and memory, as well as error codes in disk I/O.

DEFINE [150] is an extension of FINE adapted for distributed systems which added new triggering mechanisms, namely temporal

triggering and spatial triggering using software traps, and new fault models, such as communication and intermittent faults.

Xception [24] is a fault injection tool that takes advantage of debugging and performance features of modern processors to inject faults and monitor their propagation with lower intrusiveness than most SWIFI approaches that depend on pre-runtime software modifications, software traps or running the application in trace mode. It supports multiple triggering mechanisms, including temporal, spatial and event-based (*e.g.*, on memory access), and a fault model that represents transient hardware faults.

NFTAPE [137] is a framework for fault injection that can be used in various types of systems and supports multiple fault models, including bit-flips in registers and memory, communication errors and I/O faults, and multiple fault triggers, including spatial, time-based and event-based. Its main advantage is flexibility and the fact that it can use different fault injectors.

Goofi-2 [131] is capable of performing fault injection using hardware-implemented and software-implemented techniques and emulates transient hardware faults in CPU registers and memory using single and multiple bit-flips.

Gigan [82] is a software-implemented fault injection tool capable of introducing faults in memory and CPU registers of a virtualized system as single bit-flips that are triggered using breakpoints.

LLFI [145] is a pre-runtime fault injection tool that introduces the faults at the intermediate code level using LLVM and supports injection of single bit-flips in CPU registers.

Marcello Cinque and Antonio Pecchia introduced a fault injection framework aimed at virtualized multi-core systems [31]. Their framework emulates hardware errors by modifying the values of special registers that belong to the Machine Check Architecture (MCA), in doing so they are able to evaluate the error handling mechanisms of the system.

CloudVal [107] is a framework based on NFTAPE that was developed to validate the reliability of virtualized environments. It supports emulation of soft errors and injection of faults mimicking delayed I/O operations and maintenance events. Its fault injector was implemented as a loadable kernel module and features a spatial-triggering mechanism based on breakpoints.

G-SWIFT [42] applies Durães *et al.* fault model for software faults [42] at the machine code-level, by modifying the binary executable, instead of directly at the source code, thus enabling fault injection in software components to which the source code is not available.

EDFI [54] embeds the faults in the source code of the target application using basic block cloning and selectively activates only the desired faults in runtime.

ProFIPy [36] is a programmable fault injection tool that allows the users to specify the desired fault model of software faults using a domain-specific language (DSL) and which can be provided using a software-as-a-service (SaaS) deployment model.

Of all the aforementioned tools, only CloudVal [107], Gigan [31] and the framework of Cinque and Pecchia [31] have been developed with cloud or virtualized systems in mind. Cinque and Pecchia's framework emulates errors caused by hardware faults by taking advantage of error reporting functionalities of the hardware and does not reproduce the effect of a hardware fault (*e.g.*, it does not perform bit-flip to emulate the effect of soft error that hits a CPU register). Gigan is capable of injecting faults that emulate soft errors in CPU register and memory locations inside VMs. CloudVal is implemented as a kernel module for Linux that can be loaded in a VM and used to inject faults representing soft errors and other models, such as delayed I/O operations and maintenance events. When compared to these, ucXception contains fault injection tools that can inject both inside VMs (like Gigan does) and in applications (like CloudVal does), as well as supporting a fault model capable of emulating software faults. Furthermore, ucXception has been made publicly available for anyone to use and modify.

3.4 SUMMARY

Dependability evaluation is an important task in the development and evaluation of reliable systems and fault tolerance mechanisms. In the context of dependability evaluation, fault injection has historically been used as a means to accelerate the occurrence of faults. As such, fault injection is used throughout this thesis to evaluate current cloud computing systems and uses the framework presented in this chapter to support these experimental evaluations. Using fault injection we also obtain knowledge required for designing mechanisms to improve the dependability of the cloud and validate the effectiveness of the developed fault tolerance mechanisms.

This chapter presented ucXception, a fault injection framework designed to evaluate cloud computing and virtualized systems and capable of emulating transient hardware faults and software faults in any application or even inside a VM or the hypervisor. It was designed to be flexible, easy-to-use and expandable, by dividing the various required functionality into components that can be combined and interchanged using a few lines of code, according to templates and examples that accompany the framework. Currently, it features various components for monitoring the target system, extracting information regarding the fault injection and validating the results, but the user can introduce his own components and integrate them with ucXception. It represents a contribution to the state-of-the-art since it is one of

the few publicly available fault injection frameworks that is capable of injecting in cloud computing and virtualized systems and that supports multiple fault models.

4 ON SOFT ERRORS AND CLOUD SYSTEMS

Hardware faults are the cause behind a significant percentage of failures in cloud computing deployments and will become more common in the future due to the expansion of datacenters, thus containing more hardware components, the miniaturization of microtransistor feature sizes, which when combined with an increase in transistor count leads to higher soft errors rates, and the popularization of energy saving techniques, such as dynamic voltage and frequency scaling, which dramatically increase soft error rate [26].

The virtualized architecture is composed by various layers (VM, PVM and hypervisor) that have different responsibilities, roles and powers. Each layer may be more or less susceptible to soft errors depending on the percentage of CPU time that it uses, as well as experiencing different failure modes depending on the code that executes in the layer.

This chapter reports the results of an experimental evaluation that used fault injection to emulate soft errors that affect the CPU across the various layers of a virtualized system. A total of 16 211 injections were performed across different components (VM, PVM and hypervisor), different workloads (HTTP and TPC-VMS) and different virtualization modes (HVM, PV and PVH). The objectives when designing this evaluation were:

- Identify the failure modes of a virtualized node and estimate their probability of occurrence;
- Analyze how soft errors in a certain layer can dictate the experienced failure modes and their probability of occurrence;
- Measure the effect of using a different virtualization mode on the failure modes and their probability of occurrence;
- Measure the error isolation coverage among VMs and between the hypervisor and the VMs when in the presence of a soft error;
- Analyze the impact that different workloads can have on failure modes and their probability of occurrence.

The knowledge obtained from this experimental evaluation was used to design the contributions presented in Chapters 6 and 7, which aim to improve the dependability of cloud computing systems.

4.1 METHODOLOGY

The methodology applied to the experimental evaluation consists on emulating soft errors using fault injection across the various layers that compose a virtualized node (VM, PVM and hypervisor) while a workload is being executed, as is described in this section.

4.1.1 *Experimental setup*

The experimental setup, which is depicted in Figure 10, was designed to represent a subset of a real-world cloud deployment where multiple machines, or nodes, cooperate to provide computing resources to the cloud clients. This study focuses on a single node that provides the resources required for consolidating multiples clients using virtualization and on the effect that soft errors can have on it.

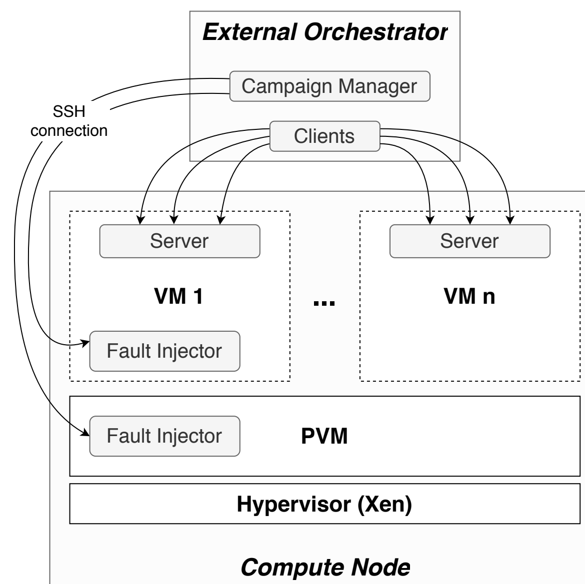


Figure 10: Experimental setup.

On this node, virtualization is provided by the Xen 4.4.1 hypervisor, which has been paired with a PVM running CentOS 7. The spawned VMs, which vary between two and three depending on the experiment being performed, use Debian 7.7 with version 3.11.1 of the Linux kernel. When two VMs are used, fault injection is performed in one of them, the so-called VM1, while the remaining VM, known as VM2, does not suffer an injection, since it has the purpose of verifying the effectiveness of the hypervisor's isolation mechanisms (*i.e.*, verify if a failure in a VM does not propagate or affect in any manner other co-located VMs). When three VMs are used, the fault is injected outside of any VM and their role is to provide a method for characterizing the impact of the fault.

Hardware-wise this node was equipped with an Intel Core i7-4770 CPU, which features 4 physical cores capable of Hyper-Threading and supporting VT-x, VT-d and EPT, along with 8GB of DDR3 RAM and two different kinds of physical storage, a 120GB SSD and a 1TB HDD.

Other than the node that is evaluated, another machine is required to orchestrate and conduct the experiments, namely, execute the workload by emulating multiple clients, trigger the fault injection and extract the results.

4.1.2 *Workload*

Two workloads were used in the experiments, an HTTP workload and the TPC-VMS workload. In conjunction both workloads cover all areas (*e.g.*, memory, disk, processing) of a system but stress different components individually.

4.1.2.1 *HTTP workload*

The HTTP workload consists of executing the Apache 2.2.22 Web server while JMeter performs multiple continuous HTTP requests, as to emulate 10 concurrent clients. Responding to each request involves computing a SHA1 hash of a 1GB array of zeros and sending the reply to the client. Since the input of the hash computation is always the same, the output must also be the same under normal conditions. A ramp-up time of 30 seconds, followed by 5 minutes of execution during which faults were injected, was configured. The load imposed by clients on the server led the CPU load to 100% during most of the experiment time. In this workload, two VMs are spawned and the load is applied equally among the two. The workload is representative of typical servers deployed in the cloud that provide services that are accessed via the HTTP protocol, where the Apache Web server is also a common choice.

4.1.2.2 *TPC-VMS workload*

The TPC-VMS [139] is a virtualization-oriented benchmark that builds upon the already existing benchmarks for non-virtualized systems (in particular, TPC-C, TPC-E, TPC-H and TPC-DS). This benchmark requires the creation of 3 VMs in the same physical server, each one running the same non-virtualization benchmark. The final results are extracted from the worst obtained value of each metric out of the three VMs, in agreement with the TPC-VMS specification. Whereas the HTTP workload was mainly CPU-intensive, with some memory and network usage, the TPC-VMS workload makes balanced use of all components of the physical system (CPU, memory, network and disk) and increases the stress on the virtualization mechanisms (more switching between VMs).

For our particular case we opted to pair TPC-VMS with the TPC-C benchmark, with the implementation being provided by jTPCC v1.2.0 [160]. jTPCC had to be modified in order to become deterministic and therefore able to evaluate the correctness of the produced output. This step required hard coding the seeds of the random number generator and limiting the number of terminals to one.

4.1.3 *Fault model*

We use the single bit-flip fault model to emulate transient hardware faults, such as those that cause *soft errors*, in microprocessor registers and main memory [121] and inject only one bit-flip per experiment run.

Injection in microprocessor registers emulates faults affecting the processor, which include faults directly affecting the registers and faults that cause errors in other processor structures (*e.g.*, arithmetic and integer unit, floating point unit, instruction decoding, internal buses, internal caches) but whose errors only have consequences when they reach the registers.

Injection in main memory emulates real faults affecting circuits (including the CPU) while a value is in transit for storage in main memory. The goal is not to emulate direct memory errors, since main memory is typically protected with error-correcting codes and secondary caches are protected with parity codes, but rather to emulate a soft error affecting unprotected circuitry (including the CPU) while a write to memory operation is in transit.

4.1.4 *Fault injection tool*

Practical fault injection was provided by the ucXception framework, using different techniques depending on the target component of injection: registers of processes running inside VMs (guest and PVM alike), memory of the hypervisor, or registers during the execution of a hypercall (hence executing in hypervisor mode). However specialized fault injection tools were built just for these experiments, which often implied modifications to Xen's source code.

When performing fault injection in processes that run inside a VM, the fault injector takes the form of a loadable kernel module for Linux, which is located in the target system and can be instantiated through a command using an SSH session or similar approaches. Once loaded, the module locates the kernel-specific data structures related to the targeted process and modifies the value of a register according to the parameters passed during the module loading. Once a context switch is made by the kernel to the specified process, that process continues execution with a corrupted register value. The registers suitable for fault injection with this tool are: *rip* (Instruction Pointer), user-space

and kernel-space *rsp* (Stack Pointer), *rax*, *rbx*, *rcx*, *rdx*, *cs*, *ss*, *rbp*, *flags*, *si*, *di*, *es*, *ds*, *fs*, *gs* and *r8* to *r15*. This tool only supports temporal triggers, hence experiments are not as repeatable as in other injectors due to timing imprecisions. Nonetheless, temporal triggers are sufficient to achieve statistical confidence of the results.

Injecting faults in the hypervisor memory is accomplished by introducing a minimalist hypercall in Xen which can be called using a user-space tool and receives the location in virtual memory that we want to target. When this hypercall is used it performs a bit-flip according to the passed parameters.

The approach that was used to inject faults during the execution of hypercalls consisted on the modification of the assembly code of Xen's hypercalls before each experiment run by adding a small assembly payload that waits for a temporal trigger (expressed as the number of CPU cycles since the computer has booted) and performs a bit-flip in our desired register. The temporal trigger is implemented by calling the "Read Time Stamp Counter" (*rdtsc*) instruction, which writes the number of CPU cycles into *rax* and *rdx*. This value is then compared to the predefined goal and if the current value is the same or higher then the injection takes place. In general, the amount of extra instructions added by this approach is minimal (only 15 instructions), which assures that the intrusiveness of the injection process can be neglected. The registers that can be targeted with this approach are *rip*, *rax*, *rbx*, *rcx*, *rdx*, *rsp*, *rbp*, *rdi*, *rsi* and *r8* to *r15*.

Figure 11 shows the assembly instructions that are added, where [register], [time], [mask] are placeholders that must be filled during compilation.

4.2 FAILURE MODES AND CLASSIFICATION

Calibration experiments, conducted along with the development of the fault injector and the experimental setup, aimed at characterizing the behavior of faulty components from the qualitative point of view. In other words, with the intent to identify the failure modes of the VMs and the hypervisor and to calibrate the tool for the remaining experiments.

In all experiments, the outcome of injecting a fault (*i.e.*, failure modes) is classified according to its impact on the output produced by the workload running inside the VM and its impact on the responsiveness of the hypervisor, which was obtained by running integrity tests at the end of each experiment.

As explained previously, two different workloads were used, an HTTP workload and TPC-VMS [139]. For the HTTP-based workload, there are five distinct failure modes, along with the possibility of an experiment having no effect on the provided service. All the failure modes represent the view from the client side (*i.e.*, the external view).

```
.globl counter
.bss
.type counter, @object
.size counter
counter:
.zero 1

pushq %rax
cmp $1, counter(%rip)
je final
pushq %rdx
rdtsc
salq $32, %rdx,
or %rax, %rdx
movabsq $[time], %rax
cmp %rdx, %rax
popq %rdx
jg final
movabsq $[mask], %rax
movb $1, counter(%rip)
xorq %rax, %[register]
final:
popq %rax
```

Figure 11: Assembly payload.

- *Incorrect content* – The application running within the VM produces syntactically correct HTML content with wrong values. Hence, incorrect content becomes visible to the service user, which may be another machine or a human. This failure mode is the most serious in its consequences, as it allows errors to propagate to other components in a system, and also the most difficult to handle at the system-level, as it is undetectable unless the output is checked against a redundant computation or using some other form of redundancy check;
- *Corrupted output* – The application produces a corrupted stream of data, while the socket remains open. The output is syntactically incorrect and the condition is detectable by clients because the server fails to comply with the HTTP protocol, sends invalid HTML code, or sends code which is not HTML at all. Consequently, the server maintains the TCP connection open but the output is corrupted. This behavior is detectable by clients and a Web browser would display an error message, as such this behavior may be considered less harmful than incorrect content, although recovery is necessary;
- *Connection reset* – The TCP connection between a client and the server is reset by the server's network stack. This corresponds

to an incorrect behavior which is, at least in part, detected by the operating system running inside a VM. To deal with such a situation, the operating system closes the socket and sends a packet with RST flag set to the client. Therefore, the connection is lost and a client rapidly detects the problem;

- *Client-side timeout* – One or more clients fail to receive a response to their request and issue a client-side timeout. The timeout is configured at 20 seconds, which is reasonably high for HTTP interactions (the keep-alive mechanism, for example, typically maintains a connection open for 5–15 seconds at the server-side). Note that numerous client requests are handled simultaneously and some responses may be correct while some are missing and lead clients to time out;

- *Hang* – The VM stops producing output and fails to answer any subsequent requests. In this case, the application running inside the VM no longer produces results and eventually all connected clients will issue a client-side timeout. A failure classified as a hang will not be classified, in our experiments, as a client-side timeout. Nevertheless, due to non-deterministic timing aspects, some experiments are classified as client-side timeouts when the actual behavior might be a hang. If a few, isolated client requests are unanswered, the failure mode is a client-side timeout; but when this occurs with many client requests, the classification may be either a hang or a client-side timeout;

- *No effect* – The injected error has no visible consequences on the service provided by any VM. Neither the performance or the correctness of the service is affected in a way that could be classified as a failure.

For each experiment, the above classification is performed by examining the output produced at every client request. For this reason, a single error may cause more than one kind of failure for the same injection. For example, the same fault may lead to a connection reset for one client request and a corrupted output for another client. Although the presentation of multiple failure modes, in a single experiment, is relatively infrequent in our experiments, some results add up to slightly more than 100% as a consequence.

The second used workload is the TPC-VMS [139] benchmark, which consists in the execution of an OLTP benchmark concurrently in 3 VMs. Unlike what happened in the HTTP workload, there is now a persistent aspect to the experiment results, which is the state of the database at the end of each experiment. For this reason, a slightly different classification scheme was used to evaluate the results of this experiment campaign.

- *Corruption* – Although the experiment terminates cleanly, when checking the final state of the database, inconsistencies are found. This is the equivalent of a Silent Data Corruption (SDC);
- *Crash* – The entire VM stops responding and the execution of the workload stops;
- *No effect* – The workload client receives every response with the expected values and, at the end of the experiment run, the database states matches the expected final state.

In either workload, in addition to the failure modes, we also tested the hypervisor integrity after each injection. To achieve this, we developed a hypervisor integrity test that consists of a battery of checks, starting by the establishment of an SSH connection to obtain a console on the hypervisor, followed by reading a file stored on the file system and running other processes (*e.g.*, *uname*, *ls*) that return deterministic values that can be used to infer the internal state of the system. A *ping* operation is also performed, but it was noted that the results from this operation cannot be completely trusted to assert the integrity of the hypervisor. We observed that a test suite like this, although simple, is comprehensive enough to accurately detect the situations where the hypervisor is unresponsive. The hypervisor integrity test leads to two possible classifications of the outcome of a single experiment.

- *Unresponsive* – The hypervisor hangs and the external probe is unable to execute the integrity tests;
- *Responsive* – After an experiment ends, the integrity tests are executed on the hypervisor successfully.

4.3 SOFT ERRORS OCCURRING INSIDE A VM

Injections inside a VM should represent the majority of real-world soft errors, given that this is the layer that accounts for the majority of CPU time and, thus, the most susceptible to this kind of faults. We subdivided our injections inside a VM into two groups:

- *Faults injected in application processes* – to characterize the failure modes exhibited to external users of the system and to determine if errors originating within one VM are able to propagate to other VMs or to the hypervisor. In other words, the application client-server interaction results in information flow which allows errors to propagate to the client. However, the hypervisor and other co-located VMs should remain unaffected by errors originating in one VM;
- *Faults injected in guest operating system processes* – aiming to examine the failure modes exhibited to external users, considering

that an error directly affecting a guest operating system has a greater potential to cause failures. Similarly to the previous category of experiments, the virtualization platform is designed to prevent such errors from propagating to the hypervisor and other co-located VMs.

4.3.1 *HTTP workload*

The results of fault injection targeting Apache processes, within one VM, for the three modes, are shown in Table 4 [1]. The results refer only to the VM where fault injection was performed, because both the hypervisor and the other VM were never affected (*i.e.*, isolation has been kept). As to facilitate comparison between the three modes, the binomial (Clopper-Pearson) confidence interval at the 95% confidence level is included next to the absolute failure counts.

FAILURE MODE	HVM		PV		PVH	
Incorrect Content	4	[0.1%; 1.0%]	6	[0.2%; 1.3%]	24	[1.3%; 3.0%]
Corrupted Output	2	[0.0%; 0.7%]	1	[0.0%; 0.6%]	1	[0.0%; 0.5%]
Connection Reset	12	[0.06%; 0.2%]	8	[0.4%; 1.6%]	1	[0.0%; 0.5%]
Client-side Timeout	130	[10.7%; 14.8%]	71	[5.8%; 9.2%]	150	[11.0%; 14.9%]
Hang	5	[0.2%; 1.1%]	6	[0.2%; 1.3%]	4	[0.1%; 0.9%]
No Effect	876	[83.0%; 87.4%]	876	[88.5%; 92.3%]	985	[82.3%; 86.6%]
Total	1027		968		1165	

Table 4: Outcomes of fault injection in application processes within a VM, HTTP workload.

We can observe that the faulty VM (*i.e.*, the one in which the faults were injected) exhibits diverse failure modes, including a large proportion of client-side timeouts and a small proportion distributed across the remaining failure modes.

In the HVM mode, about 85.3% of the injected faults had no effect on the target VM. From all the injections using this virtualization mode, only 4 (0.4%) caused incorrect content to be sent in response to

1 The sum of failures in HVM mode amount to more than 100% of the injected faults, because in some occasions a fault caused more than one failure mode.

client requests. These faults caused transient failures at the server-side, as the server continues working correctly after sending the incorrect content to the client, but with a potentially permanent effect on the client, as the values received from the server are incorrect. This is a relevant result, as these are exactly the most dangerous type of faults, as they cause the application to produce incorrect results without being noticed, unless there are explicit redundant calculation or any form of data error mechanism over the application output. In the prospect of the increase of the soft error rate in forthcoming generations of hardware, this is precisely one of the types of consequences that must be avoided or mitigated. Even so, the observed number of faults that caused incorrect content can be considered small (0.4%), but this judgment is, of course, dependent on the criticality of the application.

When comparing the results between the three virtualization modes the failure modes of the VM in which bit-flips were injected are similarly distributed, yet with some small variations. Perhaps the most noticeable differences are the higher percentage of incorrect content that was seen in PVH (+1.7%) and the considerably higher proportion of non-effective injections in PV mode (+5.2%). Hence, although the set of possible failure modes exhibited by a VM running in any of these modes are the same, the exact proportions differ slightly.

Fault injection in processes of the operating system, within one VM, was also conducted with the objective of understanding the effects of soft errors affecting one VM, determining if the error isolation provided by the hypervisor is adequate and examining possible differences between virtualization modes. The results are summarized in Table 5.

These results indicate that bit-flips in operating system processes are less likely to affect the application service than errors injected directly in application processes. It is also worth noting that every fault with impact had a manifestation as client-side timeouts or VM hangs. These results do not necessarily exclude the possibility of a soft error in a guest operating system leading the application software to produce incorrect content, but provide some evidence that such errors are much less likely (compared to injections directly affecting application processes).

Along with the failure modes, understanding error manifestation with respect to latency and duration is particularly relevant to the design of appropriate fault tolerance mechanisms. To this end, we analyzed a subset of the results presented in this section with regards to the time between fault injection and the first manifestation resulting from its injection. This analysis was conducted for faults injected in user-space applications of a VM, more specifically in Apache processes during the HTTP workload, and excluded those that had resulted in hangs and client-side timeouts. Client-side timeouts are configurable and have the value set to 20 seconds in our experiments; hangs are

FAILURE MODE	HVM		PV		PVH	
Incorrect Content	0	(0.0%)	0	(+0.0%)	0	(+0.0%)
Corrupted Output	0	(0.0%)	0	(+0.0%)	0	(+0.0%)
Connection Reset	0	(0.0%)	0	(+0.0%)	0	(+0.0%)
Client-side Timeout	5	(1.0%)	2	(−0.1%)	4	(−0.2%)
Hang	4	(0.8%)	1	(−0.4%)	3	(−0.2%)
No Effect	493	(98.2%)	225	(+0.5%)	500	(+0.4%)
Total	502		228		507	

Table 5: Outcomes of fault injection in OS processes within a VM, HTTP workload.

classified at the end of each experiment; error manifestation latency in such cases would therefore provide meaningless values.

Figure 12 shows the error manifestation latency distribution, for the HVM and PV modes (values stacked on top of each other). The x-axis shows time using a logarithmic scale. There are a few clusters of errors, and one may observe that many errors remain dormant for little over 10 seconds before the first manifestation. One extreme outlier had an error manifestation latency of 326 seconds. These time values have a potential impact on the design of error-recovery mechanisms, given that backward recovery techniques are designed to eliminate errors, and may consequently require lengthy rollbacks.

Duration of erroneous behavior is also an important concern when designing error-handling mechanisms and, more specifically, the mechanisms that should handle the errors when they are detected. Figure 13 summarizes the duration of such events, that is, the number of requests that were affected by a single bit-flip. It is worth noting that in the present case there are no specific error detection mechanism installed that could mitigate or recover from the erroneous behavior exhibited after the injection of the fault.

The temporal duration, between the first manifestation and the last manifestation of an error, was always below 1 second (the granularity with which we conducted the analysis). For this reason, we use the number of requests affected by a single bit-flip on the x-axis of Figure 13.

Excluding hangs (which have a permanent duration), the majority of errors affected only a single client request. Nevertheless, in the

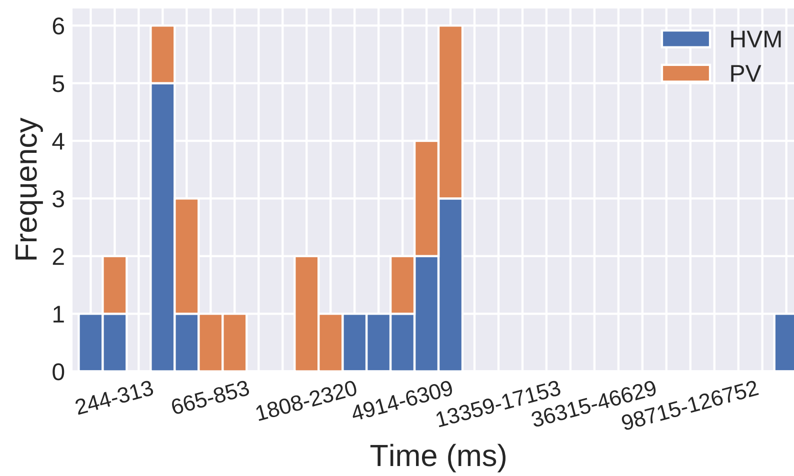


Figure 12: Manifestation latency, for effective errors, both in PV and HVM, HTTP workload.

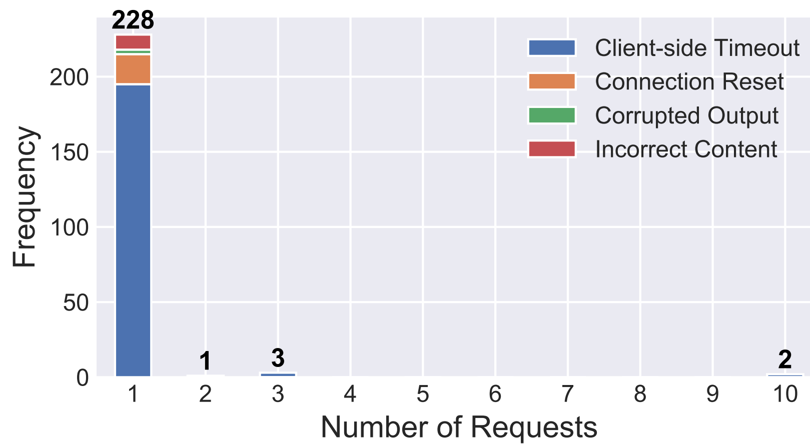


Figure 13: Number of client requests affected by a single bit-flip error, for HTTP workload.

course of our experiments, we observed that a single bit-flip error may lead up to ten client requests being unanswered, thereby leading to client-side timeout.

Figure 14 summarizes the failure modes observed for each register, for all experiments in which a fault was effective. The results in that figure concern faults injected in microprocessor registers during the execution of Apache processes. From all the targeted registers, fourteen resulted in effective injections, although the used virtualization mode lead to different subgroups of effective registers. Similarly to what many other studies have shown in the past, the Instruction Pointer (*rip*) and the Stack Pointer (*rsp*) have a high contribution to VM failure modes. The *rbx* register is also relevant, both due to the overall contribution to effective errors and the specific contribution to the incorrect content classification. The large majority of experiments that were classified as incorrect content (silent data corruptions) were caused by bit-flips in the *rbx* register. The *fs* register also had a noticeable contribution to hang failures in all modes. When comparing the three modes, PV had the lowest amount of effective registers, only seven, whereas HVM had eleven and PVH had ten. Hence, while the registers that contribute the most to failures are similar, the three modes are apparently different with respect to the root causes of failures, if we take into account also registers with a lower probability of manifestation. Furthermore, the failure modes varied occasionally between modes. One example is *r13* in PVH, which caused only corrupted output failures, and *r13* in HVM which caused only connection reset failures. According to these results, one may not conclude that some fault tolerance mechanism that works for one mode will also work for the other modes, thereby requiring some caution when designing and evaluating such mechanisms.

4.3.2 *TPC-VMS workload*

For the TPC-VMS workload, faults were injected in the various processes that supported the DBMS (PostgreSQL), namely:

- *checkpointer* – automatically performs a checkpoint, which consists in the flushing of all dirty data pages, after a certain time interval has elapsed;
- *writer* – handles the writing of data pages from shared buffers into storage;
- *autovacuum launcher* – periodically reclaims storage space being occupied by deleted or obsolete information and, then, updates the table statistics that will be used by the query planner (equivalent to calling *VACUUM* and *ANALYZE* commands);

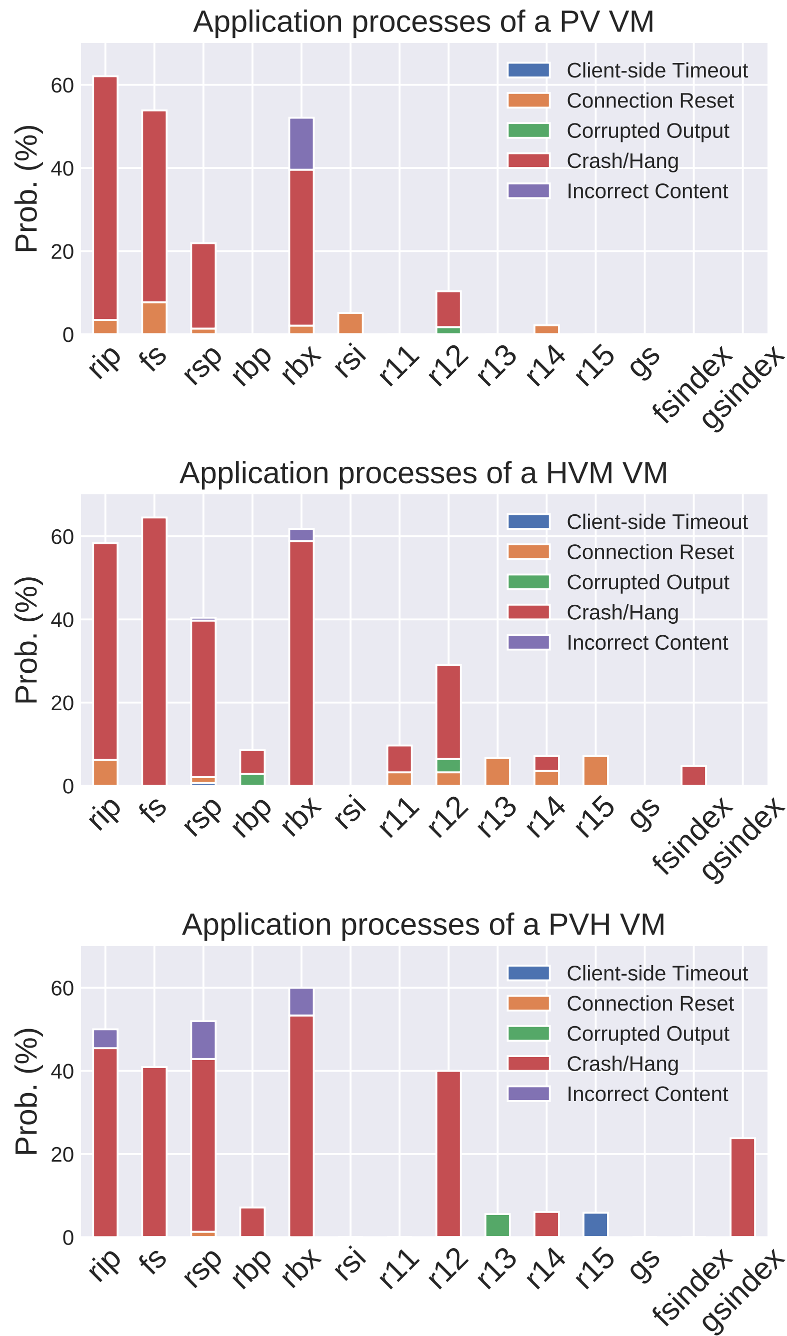


Figure 14: Distribution of failure modes across processor registers, for injections in application processes and HTTP workload.

- *stats collector* – continuously collects information, such as the number of access to tables and indexes or the number of rows in tables;
- *postmaster* – this process is always active, awaiting for connections and performing periodic tasks, even when there is no connected client.

Table 6 presents these results as a breakdown of the behavior seen in each process. For the HVM mode a total of 1080 runs were performed, while for the PV mode 2622 runs were performed.

PROCESS	HVM			PV		
	Crash	Corruption	No Effect	Crash	Corruption	No Effect
checkpointer	0.0%	18.3%	81.7%	1.3%	0.0%	98.7%
writer	0.0%	26.1%	73.9%	0.7%	22.3%	77.0%
wal writer	0.0%	20.0%	80.0%	2.6%	20.2%	77.2%
autovacuum	0.0%	18.9%	81.1%	2.5%	20.8%	76.6%
stats collector	0.0%	0.0%	100%	2.6%	0.1%	97.3%
postmaster	0.0%	20.6%	79.4%	3.3%	18.4%	78.3%
Total	0.0%	17.3%	82.7%	2.4%	12.1%	85.5%

Table 6: Outcomes of fault injection in application processes within a VM, TPC-VMS workload.

As seen with the HTTP workload, in general, between virtualization modes the experienced failure modes are the same but with slightly different proportions. A contradiction to this observation is seen in the *checkpointer* process, where database corruption only occurred in the HVM mode. Possibly indicating that the differences between virtualization modes can indeed have a significant impact in the behavior of the applications under faults but only in certain cases.

All the corrupted states caused by the *postmaster* process occurred after an abrupt DBMS crash leading to a state with missing rows, but no extraneous or incorrect rows, unlike what was sometimes seen in the other processes, where a corrupted database was missing rows, had rows with wrong content, had extraneous rows or had rows out of position.

Surprisingly, in PV mode there were several occurrences of crashed VMs, which runs against common sense that a user-space application cannot crash the VM where it is running and differs from what was seen in the HVM virtualization mode. An analysis of the hypervisor logs seems to indicate that the activation of a software bug in Xen might be the cause behind these crashes, although further research is

needed before blame can be attributed with confidence. In any case, the neighboring VMs were not affected.

These results reinforce the observation that the hypervisor correctly isolates the VMs and prevents soft error propagation among VMs or from VMs to the hypervisor. But they also confirm that there are a small percentage of faults that cause incorrect application results and corrupted state and that the errors caused by faults affecting applications running in a given VM may propagate and affect the operating system of that VM causing a hang.

Table 7 refers to injections in operating system processes while using the TPC-VMS workload, for HVM and PV virtualization modes.

FAILURE MODE	HVM		PV	
Crash	0	(0.0%)	2	(+0.3%)
Corruption	1	(0.2%)	2	(+0.1%)
No Effect	526	(99.8%)	671	(−0.4%)
Total	527		675	

Table 7: Outcomes of fault injection in OS processes within a VM, TPC-VMS workload.

Interestingly, 3 occurrences in total of corrupted state were seen, which suggests that, at least for OLTP workloads, faults in kernel-space processes can indeed cause more than just hangs and timeouts, thus reinforcing the notion that, although, no incorrect content was produced during the HTTP workload, it might still be a possibility.

4.4 SOFT ERRORS OCCURRING INSIDE THE PVM

We conducted campaigns targeting the hypervisor's PVM, known as dom0 in Xen's terminology, which is characterized by providing support to the execution of the hypervisor and having full hardware access. Hence, although hardware protection prevents it from directly corrupting Xen's ring 0 address space, PVM code can theoretically interfere with the execution of the entire system. The campaigns focused on injecting faults in the processor registers while the processor is executing code from processes related to Xen, as these are a part of the virtualization infrastructure. The targeted processes all have a specific role within the hypervisor, namely:

- *qemu* – A Qemu process is used for every HVM guest that is executing, in order to provide emulation of certain components. However, even if no HVM guest is running, there is always one Qemu instance in the system, that is used by the PVM to access image files and to provide VNC servers;

- *oxenstored* – this process implements a key-value store that holds information about the system and its VMs. Communication with this process can be done through UNIX domain sockets (PVM only), or through the kernel ring interface (*i.e.*, xenbus);
- *xenwatchdogd* – is a user-space processes that makes part of Xen's internal watchdog mechanism;
- *xenconsoled* – is a process that receives and stores in disk the logs coming from the hypervisor and guests;
- *xenbus* – this process allows inter-VM communication, particularly as a way of providing hardware access to the guests. All the device drivers running in the PVM must register themselves with this process before they can be used by other VMs;
- *xenbus-frontend* – the device drivers in Xen follow a split model design and consist in two parts, the front-end and the back-end. The front-end executes in the guest and connects to the back-end, which is running in the PVM. In the case of this process, the front-end is also running in the PVM but only to serve as the entry point for the PVM itself.

The results of the injection campaign that used the HTTP workloads are summarized in Table 8. The six processes belonging to the PVM's operating system (*oxenstored*, *xenconsoled*, *qemu*, *xenwatchdogd*, *xenbus* kernel process and *xenbus-frontend* kernel process) had different effects on the system. Of these six processes, four are user-space processes and two are kernel-space processes.

PROCESS	HANG		TIMEOUT		NO EFFECT	
qemu	34	(6.1%)	18	(3.2%)	503	(90.7%)
oxenstored	0	(0.0%)	0	(0.0%)	234	(100%)
xenwatchdogd	36	(12.4%)	31	(10.7%)	223	(76.9%)
xenconsoled	0	(0.0%)	0	(0.0%)	398	(100%)
xenbus	124	(66.0%)	64	(34.0%)	0	(0.0%)
xenbus-frontend	186	(64.5%)	100	(34.7%)	2	(0.7%)
Total	380	(19.5%)	213	(10.9%)	1360	(69.6%)

Table 8: Outcomes of fault injection in PVM, HTTP workload.

The two kernel-space processes (*xenbus* and *xenbus-frontend*) had several cases in which the system was brought down into a hanged state. Two user-space processes, running within the PVM, also had the same effect (*qemu* and *xenwatchdogd*). The other two PVM user-space processes never produced any effect on the system when targeted

with faults. This means that PVM processes, even when running in user-space, have the potential to cause the system to hang. The effect of injecting faults into a PVM process depends more on the kind of responsibility within the system, than on the execution mode.

Of all the injected faults, 30.4% caused the hypervisor to hang, remaining unresponsive and failing the integrity tests. The remaining 69.6% of the faults had no manifestation within the duration of the experiments, neither affecting the hypervisor nor any of the VMs.

In those cases where the hypervisor failed, both VMs failed as well. Furthermore, the failure modes exhibited by VMs were either hang or client-side timeout. As we described earlier, in our experiments some VM failures are classified as a client-side timeout when the behavior is also consistent with a hang. Nevertheless, we believe that the two different classifications have a common root cause, in which VMs are unable to continue executing.

A similar campaign was done with the TPC-VMS workload, where different virtualization modes (HVM and PV) were also tested in order to study if the virtualization mode used in the VMs has any impact on the outcome. Table 9 shows the results broken down per process, where 180 faults were injected in each process, for a total of 1080 faults.

PROCESS	HVM GUESTS			PV GUESTS		
	Crash	Corruption	No Effect	Crash	Corruption	No Effect
qemu	9.4%	0.0%	90.6%	8.9%	0.0%	91.1%
oxenstored	0.0%	0.0%	100%	0.0%	0.0%	100%
xenwatchdogd	23.3%	0.0%	76.7%	25.6%	0.0%	74.4%
xenconsoled	1.7%	0.0%	98.3%	5.6%	0.0%	94.4%
xenbus	100%	0.0%	0.0%	100%	0.0%	0.0%
xenbus-frontend	100%	0.0%	0.0%	100%	0.0%	0.0%
Total	39%	0.0%	61%	40%	0.0%	60%

Table 9: Outcomes of fault injection in PVM, TPC-VMS workload.

As had happened with the HTTP workload, the only failure mode is the crashing of all VMs along with the hypervisor. The kernel processes *xenbus* and *xenbus-frontend* had a very high impact, whereas the *oxenstored* process had no impact and the *xenconsoled* process had little effect. The virtualization mode of the VMs showed little to no impact on the failure modes and their proportions.

Figure 15 shows the failure percentage for faults injected in processes of the PVM with error bars at the 95% confidence level, discriminated by workload.

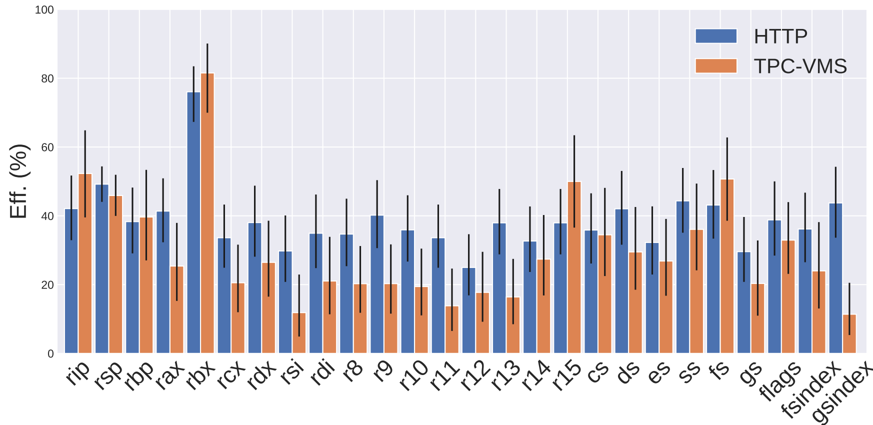


Figure 15: Failure percentage in processes of the PVM.

It is possible to observe that failures in the PVM were caused somewhat uniformly by every register apart from *rbx*, which was the most prone to causing failures. The used workload appeared to exercise relatively little effect on failure probability, likely because the operation of the studied PVM applications did not vary significantly based on workload.

4.5 SOFT ERRORS OCCURRING IN THE HYPERVISOR

In a virtualized environment, the hypervisor is responsible for implementing the essential functions (*e.g.*, allocating CPU time, managing memory and other hardware access) and isolating the VMs. It represents a single point of failure in the architecture that is associated with this type of virtualized systems.

4.5.1 *In the hypervisor memory*

As to evaluate the impact of soft errors affecting memory in a virtualized system, we conducted a campaign of injections in the memory section reserved for Xen, which consisted of 276 runs and found only one occurrence of a *Hang* of the entire infrastructure, and a campaign in the memory of the PVM, which consisted of 102 runs which yielded no failure.

The relative low number of runs, specifically when talking about memory injections, was the main contributor to almost no failures being experienced. In order to improve the efficiency of injections, a pre-injection analysis to identify the memory locations that should be targeted would further improve the results of the experiments [13].

4.5.2 *During the execution of hypercalls*

Given the size of Xen codebase we opted to restrict register bit-flips to the parts that had the highest CPU usage. In particular, we focused on hypercalls because they are called very often and provide the link between the hypervisor and the VMs. To choose between the dozens of hypercalls that Xen provides, we performed a profiling phase where both workloads were executed without any faultload, while we recorded how many times each hypercall was called. A better metric would have been the total CPU-time spent by each hypercall, however the profiler was unable to obtain this value. The results for the HTTP and TPC-VMS workloads are displayed in Table 10.

		HTTP		TPC-VMS	
ID	HYPERCALL NAME	# CALLS	%	# CALLS	%
1	mmu_update	2092	2.68	23099	0.29
3	stack_switch	10232	13.10	847445	10.73
5	fpu_taskswitch	-	-	264869	3.35
7	platform_op	-	-	1	<0.01
10	update_descriptor	-	-	957687	12.13
12	memory_op	-	-	389	<0.01
13	multicall	552	0.70	331361	4.20
14	update_va_mapping	90	0.11	15342	0.19
17	xen_version	517	0.66	18012	0.23
20	grant_table_op	-	-	218832	2.77
23	iret	31758	40.68	2471921	31.30
24	vcpu_op	17666	22.63	1256245	15.90
25	set_segment_base	2488	3.18	173044	2.19
26	mmuext_op	2157	2.76	73398	0.93
29	sched_op	6323	8.10	644621	8.16
32	evtchn_op	1458	1.86	447432	5.67
33	physdev_op	2724	3.48	152813	1.94
35	sysctl	1	<0.01	1	<0.01

Table 10: Hypercall profiling for both workloads.

With this information, the *iret* and *stack_switch* hypercalls, which account for about half of all the calls, were chosen as the targets for fault injection. The *stack_switch* hypercall is used by a VM to register the kernel stack segment and pointer into Xen, which is required because the TSS is not fully virtualized by Xen. The *iret* hypercall mimics the behavior of the *iret* instruction of x86 processors. This

hypercall returns the execution flow from the interrupt handler routine to user-space after a hardware or software interrupt.

4.5.2.1 *HTTP workload*

Table 11 and Table 12 present the results for injections during the HTTP workload, for the iret and stack_switch hypercalls respectively.

HYPERVISOR	FAILURE MODE	VM1	VM2	BOTH VMS
Unresponsive 94	Incorrect content	0	0	100%
	Client-side timeout	0	0	
	Hang	94	94	
	No effect	0	0	—
Responsive 2082	Incorrect content	60	48	74.1%
	Client-side timeout	13	7	
	Hang	348	351	
	No effect	1661	1676	—

Table 11: Outcomes of faults affecting the iret hypercall of Xen, HTTP workload.

HYPERVISOR	FAILURE MODE	VM1	VM2	BOTH VMS
Unresponsive 28	Incorrect content	0	0	100%
	Client-side timeout	0	0	
	Hang	28	28	
	No effect	0	0	—
Responsive 2001	Incorrect content	0	0	83.4%
	Client-side timeout	1	37	
	Hang	732	811	
	No effect	1268	1153	—

Table 12: Outcomes of faults affecting the stack_switch hypercall of Xen, HTTP workload.

Of particular importance is the high percentage of incorrect content failures when injecting in the *iret* hypercall, whereas no occurrences of this failure mode happened in the *stack_switch* hypercall. It shows that each hypercall causes different failure modes that can have different impact, therefore prioritization can be done when designing fault tolerance mechanisms for hypercalls. Another observation is that injections in hypercalls can affect either VM individually or both VMs at the same time.

4.5.2.2 *TPC-VMS workload*

The same hypercalls were targeted when executing the TPC-VMS workload, with the results presented in Table 13 and Table 14.

HYPERVISOR	FAILURE MODE	VM1	VM2	VM3	2 VMS	ALL VMS
Unresponsive 419	Crash	0	0	0	—	419
	Corrupted State	0	0	0	—	—
	No effect			0		
Responsive 860	Crash	21	15	5	3	0
	Corrupted State	0	0	0	1	0
	No effect			815		

Table 13: Outcomes of faults affecting the iret hypercall of Xen, TPC-VMS workload.

HYPERVISOR	FAILURE MODE	VM1	VM2	VM3	2 VMS	ALL VMS
Unresponsive 342	Crash	0	0	1	—	340
	Corrupted State	0	1	0	—	—
	No effect			0		
Responsive 1257	Crash	1	3	5	0	0
	Corrupted State	4	7	4	1	0
	No effect			1232		

Table 14: Outcomes of faults affecting the stack_switch hypercall, TPC-VMS workload.

In both hypercalls, there were occasions where two out of the three VMs suffered a failure caused by a single bit-flip, which proves that unlike what was seen in fault injection campaigns that targeted processes in the VM or the PVM, a single fault can affect a combination of VMs other than only one or all the VMs. By using the TPC-VMS workload, which keeps persistent storage, we found that certain faults were able to cause corrupted state in the database that was imperceptible to the client (*i.e.*, the DBMS continues to provide service to the client, but using incorrect content). When comparing both hypercalls, the *stack_switch* hypercall appears to be more likely to cause a crash and less prone to lead to a corrupted state than *iret*.

In the *stack_switch* hypercall, there was one fault that caused two out of the three VMs to end with corrupted state, thereby showing that a single fault affecting one hypercall can silently corrupt data in various VMs simultaneously. More importantly, it proves that there is a risk in using redundant VMs in the same physical host as a mechanism against soft errors (*e.g.*, run the same workload, feed the same inputs

and then compare the produced output), because they can be affected in the same way and return the same incorrect output.

Another new observation is that, as occurred with the *iret* hypercall, the hypervisor can be classified as unresponsive, but its VMs can continue to operate, although with failures.

4.5.2.3 *Detailed analysis*

Supported by the availability of the assembly code of the hypercalls, a more detailed analysis was performed to explain why certain failure modes occur and which variables affect the failure modes and probabilities of this component.

Figure 16 presents four heatmaps that reflect how register and fault location (*i.e.*, the line of assembly code where the fault is injected) affect the percentage of faults that lead to failures, for each workload and hypercall. Darker colors represent higher failure probability than lighter colors.

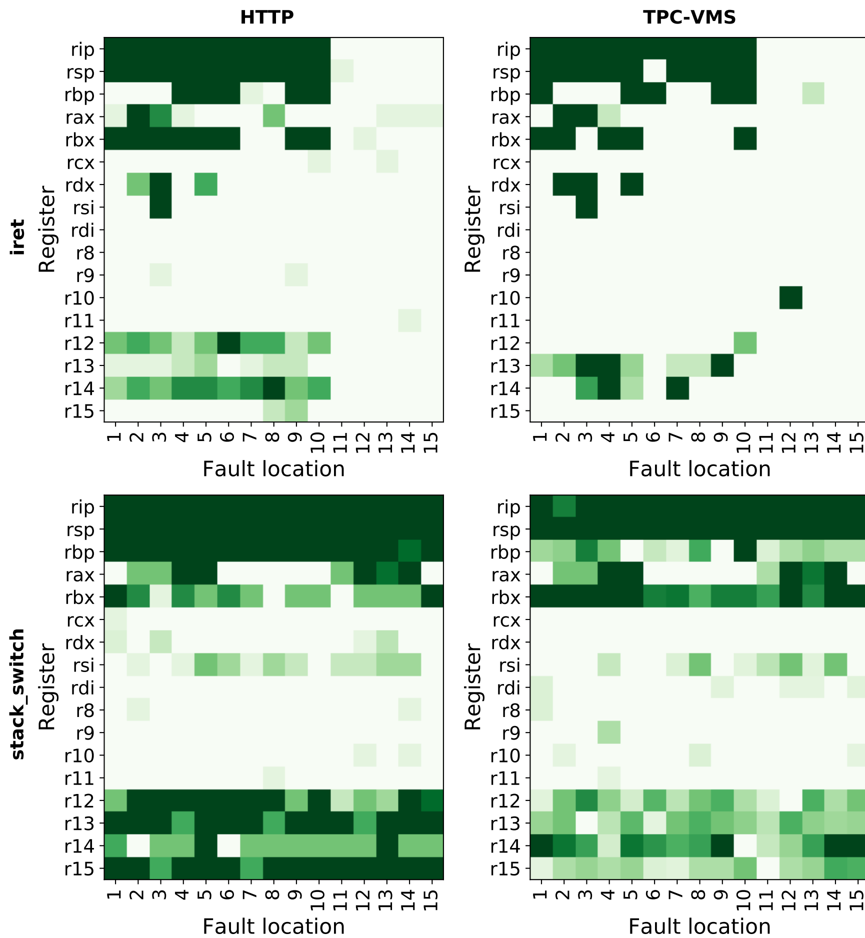


Figure 16: Heatmap of failure probability for each register/fault location pair.

Immediately it can be understood that different workloads and hypercalls possess slightly different failure percentages. While the *stack_switch* hypercall had homogeneous performance across its locations, the *iret* hypercall had almost no failures in its later locations. This is explained by *stack_switch* being a very compact function, whereas *iret* is bigger and contains error handling code at the end. Hence most of the later locations of *iret* fall on this error handling portion, which is rarely activated.

In terms of the impact that individual registers have on failure probability, we see that the *rip* and *rsp* registers consistently have very high failure probability, although it is expected given their importance in the x86 architecture. Registers *rbp* and *rbx* also had a tendency to cause a large percentage of failures, however the magnitude of their effect varied depending on the hypercall. An analysis of the assembly code showed that both hypercalls had different levels of usage of *rbp* and *rbx*, which explains this observation. Namely, while *stack_switch* did not save the value of *rbp* in the stack for restoring it later (probably due to a compiler optimization), *iret* did, hence *rbp* causing more failures in *stack_switch* than in *iret*. On the other hand, *stack_switch* used a less varied range of registers while *iret* used more registers and made particularly intensive usage of *rbx* for indirect memory addressing. Interestingly, registers *r12* to *r15* had significantly large probability of failure, despite not being used directly by any of the hypercalls. This suggests that the failures caused by these registers were long latency failures, where the bit-flip propagated out of the hypercall and affected other functions.

A similar analysis was performed wrt. silent data corruptions, the results show that registers *r12*, *r13* and *r14* (those that cause long latency failures) were the main culprits, followed by the *rbx* and *rbp* registers. Interestingly, different workloads had a strong impact in which registers caused SDC even in the same hypercall. A possible justification is that since most failures occur in the functions that call the hypercalls and not the hypercalls themselves, different workloads lead to different functions being used.

A sample of runs that showed interesting failure modes was manually analyzed to understand why those failures occurred. We will refer to them as runs `A` to `F`.

Run `A` refers to an injection in the *rsp* register, bit 60 and location n° 7 of the *iret* hypercall when running TPC-VMS. It resulted in a crash of all VMs and hypervisor. In fact the hypervisor restarted automatically due to this fault. The explanation for this failure is based on the fact that the *rsp* register is used as a basis for memory addressing just a few instructions after the bit-flip. If the value in *rsp* points to an invalid memory area then the hardware will throw an exception and the system is restarted. The large majority of full VM crashes follows this pattern.

Run B is a case where one VM sent a single incorrect response to the client during the HTTP workload. This response maintained the HTML structure but lacked the SHA1 hash. The injection that lead to this behaviour affected the *rbx* register, bit 21 and location nº 5 of *iret*. An analysis of the assembly code showed that the value in *rbx* is used immediately after the bit-flip to specify the destination of a memory write. We can imagine that the value of *rbx* was still valid and the write operation affected a memory area assigned to the 'sha1sum' process of a VM, whose job is to calculate an hash for each response. This may occur because Xen is capable of writing in any memory area that it wants, even if it belongs to a VM. Consequently, the 'sha1sum' process possibly crashed due to its memory area having been corrupted. However, one can imagine that in some very rare occasions, the written value could be benign enough not to crash the process but still enough to cause the process to produce an incorrect hash.

Run C is another case where a VM sent a response without hash to the client. This time the affected register was *rdx*, bit 50 and location nº 5 of *iret*. From analyzing the assembly code we see that the value of *rdx* is stored in the stack and then overwritten. Hence the corrupted value propagates through the stack. However we were unable to further follow its path.

Run D is a case where a VM stopped responding but was still alive as shown by Xen toolstack. This resembles a 'hang' failure of the VM, but we are not able to completely confirm it *a posteriori*. In most other situations of single VM crash, the VM crashes completely and does not appear as running when using the toolstack. This behaviour was caused by injecting a fault in register *rbp*, location nº 13 and bit 13 of the *stack_switch* hypercall during the HTTP workload. The value of *rbp* is not used in *stack_switch* but it is also not cleared or restored from stack (as happens in *iret*). We can imagine that a different *rbp* value changes the return address used by a 'ret' instruction (called to return from *stack_switch*), which will change the execution flow unpredictably.

Run E refers to an injection in *stack_switch* when running the TPC-VMS workload. The target register was *rbp* and the fault affected bit 59 on location nº 8. The same explanation given for run D can be applied here, despite the experienced failure mode being different.

Run F is a case where silent data corruption was found in the database state of two VMs after executing TPC-VMS. VM1 had 4 rows with wrong numeric values and 2 missing rows, whereas VM2 had 20 rows with wrong numeric values, a couple of rows out of position and 12 missing rows. VM3 completed the workload correctly. The fault was injected in *rbp*, bit 49, location nº 1 of *iret*. Since this bit-flip takes place immediately before *rbp* is pushed to the stack, which will only be popped before returning from the hypercall, it appears that the

corrupted *rbp* leads to a different return address being used which changes the execution flow.

4.6 RELATED WORK

Fault injection has been employed in the past to evaluate virtualized and cloud computing systems, although with different scopes and objectives from those of the present study. M. Le *et al.* had some of the first works [81, 82] to combine fault injection with virtualization. Their research had the objectives of understanding how virtualization could be applied as a mechanism to facilitate fault injection and the challenges associated with performing fault injection through virtualization. To this end they developed a fault injection tool that would eventually be called as Gigan and performed fault injection campaigns using multiple fault models, which included single bit-flips in random memory locations and in a few CPU registers paired with a location-based triggering mechanism that was applied to the most often called functions, across VMs, PVMs and the hypervisor. The results showed that fault injection using virtualization can produce similarly accurate results as SWIFI performed at other layers (*e.g.*, by the operating system) and even found several instances where the evaluated hypervisor (Xen) had allowed isolation violations between VMs.

Pham *et al.* proposed CloudVal [107], a framework for performing fault injection-based experiments for assessing the reliability of virtualized environments. The frameworks supports not only the traditional single bit-flip in memory and CPU registers, which is representative of the effect of soft errors, but also includes more artificial fault models that emulate guest misbehaviour and performance faults.

Xin Xu and H. Howie Huang evaluated the propagation of soft errors inside a Xen hypervisor with recourse to fault injection in a simulated environment [156, 157] and concluded that a soft error may propagate between CPUs through shared hypervisor data structures, as well as it may also propagate from the hypervisor to a PVM or a VM, and that there might be significant latency between the moment of the soft error and the moment when the failure occurs.

Marcello Cinque and Antonio Pecchia [31] have developed a framework for performing fault injection in virtualized multicore systems, aimed at evaluating critical embedded systems, that emulates hardware faults through the machine check architecture.

4.7 SUMMARY

Hardware faults, such as those that cause soft errors, will pose an increasingly bigger problem to the dependability of cloud computing due to the ever increasing amount of hardware that constitutes the

global cloud computing infrastructure and the natural tendency for improvements in microprocessor manufacturing and energy saving techniques. An experimental evaluation with recourse to fault injection of soft errors was performed in a virtualized system, similar to those that can be found in cloud computing. Injections in the various layers of a virtualized system (*i.e.*, the VM, the PVM and the hypervisor) and across different workloads were performed. Despite a small number of previous research works having focused on evaluating the dependability of virtualized or cloud computing systems, an evaluation that covered a real cloud computing node, instead of a simulation, and which performed injection on the various layers of such a system was absent from the literature before the current work was produced. This work yielded various key observations with regards to failure modes and probabilities, which can be summarized as follows:

- Soft errors that affect application processes of a VM may cause localized failures, which do not propagate to other VMs in the system. Such failures include client-side timeouts, crashes of the service and failures where incorrect content is stored locally or transmitted to the service clients;
- Soft errors occurring inside the PVM have either no effect or cause the entire virtualized system to crash;
- Soft errors during hypercalls (*i.e.*, representative of hypervisor execution) can affect one or multiple VMs, including all the VMs, in the system, causing incorrect data as well as service crashes.

Thus we can conclude that, while the *isolation provided by the hypervisor is effective* in preventing a soft error from propagating from a VM to another, *common-mode failures* that affect more than one VM and can lead to entire crashes of the virtualized system and even to incorrect content being produced by multiple VMs are a possible consequence of soft errors that affect the components of a virtualized architecture. Even though internal isolation is maintained when soft errors occur inside a VM, failures, which include hard-to-handle silent data corruption, can propagate to service clients via information flow. Moreover, the fact that more than one VM can produce incorrect content due to a single soft error also means that using redundancy mechanisms that depend on different replicas running on the same physical virtualized host is strongly discouraged, as it may suffer from common-mode failures that cause the mechanism to fail. Although the virtualized system may crash along with its VMs, this does not mean that the state of the VMs becomes corrupted (as will be verified in Chapter 6), but only that the hypervisor or PVM, which are a single point of failure of the virtualized architecture, have stopped working correctly.

Other observations that better detail the behaviour of a virtualization system and merit enumeration are:

- Different virtualization modes of the VMs show, in general, similar failure modes and probabilities;
- Inside a VM, the most common failure mode leads to service unavailability (client-side timeouts, crashes or hangs);
- Inside a VM, soft errors in application processes are more likely to lead to client-visible failures than kernel-space processes;
- In rare occasions when injecting on hypercalls, the correctness test classified the hypervisor as unresponsive however its VMs continued to execute nevertheless;
- The two tested hypercalls showed different failure mode distribution.

Fault tolerance mechanisms capable of handling the identified failure modes must be able to deal not only with the most common failure mode (*i.e.*, service crashes), but also with failures that cause dangerous silent data corruption. Whereas timeouts, hangs and crashes can be easily dealt with using well-established approaches, such as timeout and retry for idempotent operations, but which ultimately depend on support from the client side, failures that cause silent data corruption require application and domain-specific error detection and tolerance mechanisms. As an example, checkpointing and rollbacks may be used to restore application or VM state to a previous and error-free copy, as long as the failure can be detected in an adequate timeframe.

Generic approaches to be applied to the virtualized node independently of what application is being executed in the VMs must be able to work around the single point of failure that is the PVM and hypervisor. Such can be attained by resorting to external redundant copies, which carries significant performance and cost overhead, or by duplicating state (*e.g.*, alternating across two hypervisor instances), hence reducing the dependency on a single component to provide virtualization.

5 ON SOFTWARE FAULTS AND CLOUD SYSTEMS

Software faults are recognized as one of the most common causes of failures, having an even bigger preponderance than hardware faults [60, 61, 101]. Cloud computing, like any software-based system, is susceptible to failures due to software faults that are not detected during the testing phase of software development. However, when compared with traditional systems, cloud computing is more exposed to the impact of software faults because it depends on software that is required for providing virtualization (*e.g.*, the hypervisor and its toolstack) or to manage the resources of the cloud (*e.g.*, cloud computing management platform) [23].

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."
- Edsger W. Dijkstra, The Humble Programmer. 1972 ACM Turing Award Lecture

This chapter presents the results of an experimental evaluation using fault injection of realistic software faults on two important software components that compose a virtualized system, more specifically, the device drivers and the toolstack. In a virtualized system it is often the case that all VMs share the device drivers that are hosted in the PVM, which has less restricted access to the hardware. Furthermore, device drivers are recognized from literature in operating systems as some of the components with the highest amount of software faults per lines of code [30] and which require tailored fault tolerance mechanisms [138]. For these reasons, device drivers represent a single point-of-failure that is prone to fail due to software faults and affect the entire virtualized system.

The toolstack consists on the set of applications and libraries that provide or facilitate the connection between the userland applications and the hypervisor. They are usually employed by a system administrator to manage the node and perform operations such as spawning or killing a VM.

These two components were chosen as targets for injection, as opposed to other components such as the hypervisor itself, because they represent new or essential software components of a virtualized system that, usually, do not receive the highest amount of testing, thus are more likely to have a higher amount of latent software faults.

Incidentally, both components reside in the PVM, *i.e.*, the VM that is responsible for managing the guest VMs, multiplexing access to the hardware, interacting with the hypervisor. Given its role and position in the virtualized system, the PVM can be viewed as a single point-of-failure. In this situation, the cloud client is unable to choose or change the software that is executed on the PVM, as it is managed by the cloud provider.

The objective set when designing this evaluation was to *identify the failure modes and estimate their probability of occurrence in a virtualized node that is affected by a software fault* and to *measure the impact that software faults in device drivers and in the toolstack can have in a virtualized system*. The knowledge obtained from this evaluation is used to design and evaluate the contributions described in Chapters 6 and 7. This chapter begins with a description of the methodology, including information about the experimental setup, workload and fault injection process, and proceeds to present and analyze the obtained results.

5.1 METHODOLOGY

Fault injection of software faults is the basis of the methodology used in this chapter. Two different workloads were used over a total of four VMs in the system and optimizations to the fault injection process were performed to speed up the campaign execution. A more detailed description of the components that served as the target for fault injection and their role in a virtualized system, the used workloads and which process was employed for performing fault injection is described in this section.

5.1.1 *Architecture & target components*

The components chosen as the targets for injection were the default toolstack of recent Xen versions (*libxl*) plus the respective command-line application (*xl*) and an Intel Ethernet driver included in the Linux kernel (*e1000e*), which supports the virtual bridge shared between all the VMs in our system (the driver can vary according to the used hardware). In the overall picture of a virtualized system, these components represent a small yet critical part.

Figure 17 compiles information extracted from multiple sources [29, 88], including the source code of the Xen project, and depicts a Xen-based virtualized system and the key components of a PVM and their interactions.

The PVM is running an off-the-shelf operating system from where it obtains the *OS Kernel* and *"real" device drivers*. Along with the *"real" device drivers*, the *back drivers* (or backend drivers) must also be provided by the operating system package. These backend drivers are characteristic to the PVM in Xen deployments and are a part of the Xen split device driver model [29, 50], which divides a device driver into two sides – the frontend and the backend. The frontend driver must be present in all paravirtualized guests, while the backend is executed only in the PVM. Their purpose is to support the interaction between the VMs and the PVM, most often to request hardware resources. This model abstracts the details about the underlying hardware from the VMs, which are now left to the *"real" device drivers* of the PVM

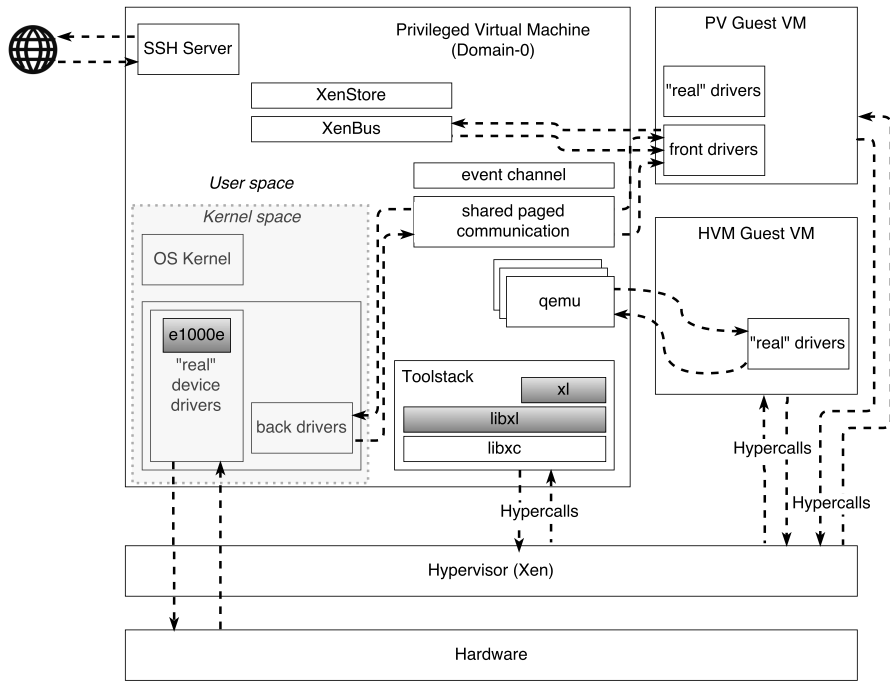


Figure 17: Architecture of a virtualized system, focusing on the PVM.

to handle, and allows the VMs to request hardware resources in a transparent manner.

The messages between the front and back side of these drivers are transmitted through *shared paged communication*, which is nothing more than reserved shared memory. This communication bus is paired with the *event channel* which serves for the hypervisor to communicate events (*e.g.*, interrupts) to all VMs.

Each PV VM requires *front drivers*, which are used by the *"real" drivers* to provide an homogeneous and transparent interface to the system's hardware. The *front drivers* will communicate the desired actions to the *back drivers* running in the PVM, which will perform those actions (through the *"real" device drivers*) and then return the results using the inverse path.

HVM guests do not require these special drivers, because thanks to hardware extensions they can access the hardware almost directly, just being restricted by those same extensions, which limit the operations available to these VMs. However, certain hardware devices are not supported by these hardware extensions and must be emulated. To do so, the PVM executes a *qemu* process for each HVM machine.

When a VM needs to communicate with the hypervisor, it must do so through hypercalls (a concept similar to system calls). However a deep level of abstraction provided by a range of libraries facilitates this process. These libraries are usually referred to as the *toolstack* and, ultimately, enable the administrator to manage the system, create and destroy new VMs, and other similar operations. At the bottom and closest to the hypervisor, we have *libxc*, which is a very low-level C

library that implements several commonly used operations. The other higher level libraries (*e.g.*, *libxl*) usually resort to *libxc* to facilitate their job and avoid duplicated code. The counterpart of the *libxl* library is the *xl* application that can be used by the administrator to take advantage of the functions implement in *libxl*.

Previous versions of Xen used different toolstacks and it is not uncommon for system administrators to change the toolstack. In fact, the usage of libraries to communicate with the hypervisor is just a user-friendly option and any user space application running in any VM is free to explicitly perform a hypercall, as the hypervisor does not limit this aspect.

A component that has a similar purpose to the *shared paged communication* is the *XenBus*, however it sees limited usage nowadays. Currently, the *XenBus* is mainly used by guests to share configuration details with the *XenStore*. The *XenStore* is a key component which is used to store configuration details for each VM and is able of setting callbacks that are triggered whenever certain fields are altered. Finally, when the cloud administrator needs to access the PVM, he usually does so by accessing the *SSH Server*.

Taking the above figure and explanation into account, it is easy to see the relative place and role of our target applications – *libxl/xl* and *e1000e*. The *libxl/xl* is expressly depicted in the figure, where it belongs to the *Toolstack* group. It is an important piece of the system and must support important and regularly used operations, such as creating new VMs. Furthermore, the administrator of the system can choose to change the toolstack used in the system with another that he is more comfortable with, which might lead to the usage of software that has not been thoroughly tested. Current versions of Xen use *libxl/xl* as the default toolstack, but in the past the *xend/xm* toolstack was the default and other toolstacks such as *libvirt/virsh* can be used.

A characterization of both components, in terms of software metrics, is presented in Table 15.

METRICS		LIBXL/XL	E1000E
Nº of files	C files	63	10
	Headers	22	3
Nº of lines	Blank	9025	3622
	Comment	9134	6190
	Code	41350	16523
Nº of functions	–	1454	485

Table 15: Software metrics of the target components.

A total of four VMs executing two different workloads concurrently were used during the experimental evaluation, supported by

the Xen hypervisor version 4.6.1. In order to cover the various available virtualization modes – paravirtualization (PV), hardware-assisted virtualization (HVM), and hybrid virtualization (PVH) – each mode was used at least in one of the four VMs (PV was used twice). The VMs run Debian 7.7 with kernel 3.18.27 and 500 Mb of RAM. The PVM (dom-0) runs CentOS 7, with a manually compiled 3.11.2 kernel. Figure 18 depicts the experimental setup.

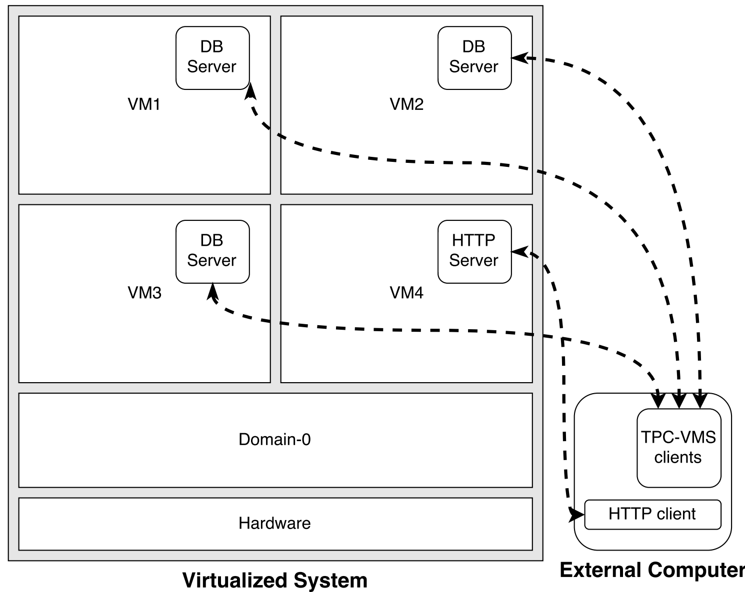


Figure 18: Model of the experimental setup.

5.1.2 *Workload*

Two workloads that represent traditional applications found in cloud computing deployments were executed simultaneously as to exercise different areas (*i.e.*, disk, networking, memory, computing) and code paths of the virtualized system. These workloads are the same that have been used in the experimental campaigns of Chapter 4.

The first workload emulates a simple, yet effective HTTP-based client-server scenario where an Apache 2.2.22 webserver serves a webpage that contains solely the results from running a SHA-1 calculation using the same input for every request. Since the input value of the hash calculation is defined *a priori* and remains unchanged, it becomes trivial to evaluate whether an injected software fault had an effect in the output sent to the client. This workload was implemented with the aid of JMeter 2.12 to emulate 10 concurrent clients performing requests in a quick fashion.

The second workload is the TPC-VMS benchmark [139], which was designed specifically for evaluating virtualized systems and consists in executing three VMs side-by-side running one of four other TPC benchmarks – TPC-C, TPC-E, TPC-H or TPC-DS. For our specific

usage, we opted by the TPC-C benchmark implemented client-side using jTPCC 1.2.0 [160] and supported server-side by the PostgreSQL 9.4.5 database server.

While the TPC-VMS benchmark reflects an I/O-heavy usage and keeps a permanent application state which can become irrevocably corrupted due to software faults, the HTTP workload is more computation-heavy and does not keep permanent state in disk. In both workloads, the first step consists in launching the required VMs and the last consists in shutting them down.

The coverage attained by both workloads, in terms of covered lines of code (LoC), has been found to be 24% for *libxl/xl* and around 30% for *e1000e*. It is debatable whether this value can be considered good for the purpose of our study, or whether a more complex workload should have been used in order to attain higher coverage percentage. There are two key reasons that make the effort of building a more complex workload both difficult and misguided. First of all, given the need of both applications to support different architectures and hardware, it would be impossible to obtain very high coverage values because certain code paths that depend on these factors cannot be taken. Second, and perhaps more importantly, the criteria behind the choice of workloads to be used should not have the objective of fulfilling a certain metric such as line coverage, but rather of representing with high fidelity the kind of workloads seen in the real world, and hence reproducing also real software faults that can affect those same systems. For this purpose we are confident of the adequacy of the chosen workloads.

5.1.3 *Fault injection technique & faultload*

We resorted to the fault injection tool capable of introducing software faults [104] that is available as part of the ucXception suite (refer to Chapter 3 for more information). This tool implements the fault model proposed by Durães *et al.* [42], which originates from a study of software faults found in real-world open-source C applications and is composed by a set of operators and associated constraints that define where and how to inject a software fault.

The process of fault injection was improved with techniques that accelerated the execution of the campaigns and improved the quality of the obtained results. Namely, the set of software faults to be injected was sorted according to the McCabe complexity number [91] of the function where each fault would be applied, so that it would be possible to obtain a decent estimate (with a global deviation $<=$ 1%) on the failure modes and probabilities using solely ~25% of all possible software faults [35]. Furthermore, a code execution profile was obtained in order to understand which lines were never executed for our specific workload, thus allowing all software faults that reside

in non-executed lines to be automatically classified as having had no effect (a software fault in a line that is never executed will never be activated). Finally, functional test suites were used, when available, or simple test suites were manually created, to be executed after the software fault has been injected but before the workload has started, as to ensure that all injected faults represent realistic faults that may occur, as we believe that if simple testing can detect the fault, then it would have been a fault that would not reach production code [98]. If the test suites fails (*i.e.*, detect the fault) then the workload is not started and the fault is classified as not having passed the testing phase.

The flow of a fault injection campaign is as follows. It starts with the generation of all possible software faults for the target application, which are stored as patch files. Then the target application is compiled with support for code coverage profiling and a base run (without any fault) is executed as to obtain information about which lines of code were executed. Finally, the McCabe complexity number is calculated for each function through an analysis of the source code and the generated faults are sorted according to this number.

After this point the fault injection experiments begin. Each experiment starts by checking whether the lines modified by the patch match at least one of the lines that are executed during the run. If this is the case, the experiment run can continue, otherwise the run finishes here and it is classified as 'No Effect'. Then, the patch (*i.e.*, the software fault) is applied, the code recompiled and installed. It is at this point that the functional tests are executed to filter out software faults that would not be realistic. If the functional test fails, the run is stopped and classified as such. Otherwise, the run continues and the workload is executed while relevant information (*e.g.*, the output sent to the clients) is being collected. Each experiment run takes around twelve minutes.

5.2 SOFTWARE FAULTS IN THE TOOLSTACK

Software faults were injected in the default toolstack (namely, *xl* and *libxl*) of Xen as to evaluate how the used toolstack, which can be chosen by the cloud administrator, affects the dependability of the virtualized node. A total of 7 811 faults were injected out of an overall 24 411 possible faults, or ~32% of all faults, as generated by the fault injection tool. The obtained results are presented in Figure 19.

Of all the injected faults, only 2 490 (31.8%) affected at least one line that was exercised by the workload. This observation exemplifies why residual software faults are difficult to detect during the testing phase, as even complex workloads fail to cover large portions of the code. From all the injected faults, 9.1% were detected by the manually created functional test, which exercises basic functions of the toolstack,

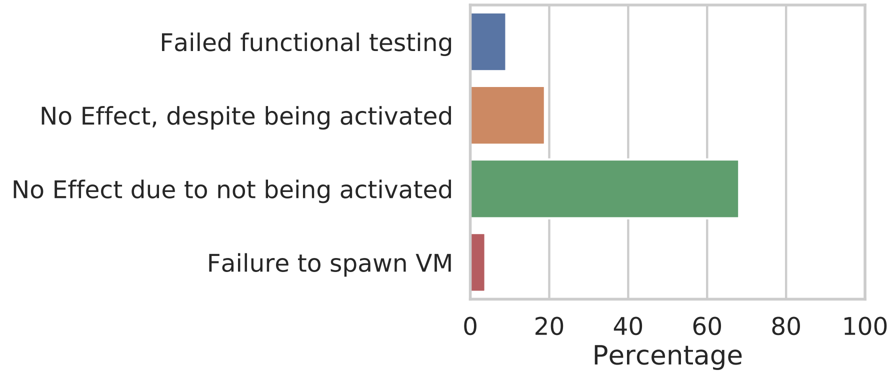


Figure 19: Failure modes and probabilities when injecting on the toolstack.

thus rendering these faults as non-representative of real software faults. The detailed analysis of the reasons that led to a fault being detected by our test suite shows that most of those occurrences were due to the *assert* [49, 119] statements introduced by the developers that detected incoherent situations and stopped the execution from going forward and possibly causing more damaging consequences. However this was not always the case and segmentation faults, attempts to free memory using corrupted pointers or entering inside of other fault handling routines were also high-noise situations that were easily detected.

Just 299 (3.8%) faults passed through the test suite and caused failures in the system, which resulted in failures to spawn the VMs that would be used for executing the workload. In these cases, the process of creating a new VM fails with an error code, but leaves behind an inconsistent state (*e.g.*, a zombie-like VM that does not run properly but appears to be alive). Obviously, more comprehensive tests would have reduced the number of such undetected faults, since they do not lead to silent failures, but this is always a difficult problem since test suites must balance coverage with execution time.

Whenever a fault affected an executed code line, passed the test suite, successfully spawned all the VMs and executed the workload, it always caused no visible effect in any VM or in the PVM, both in terms of overall availability and data corruption of output sent to the exterior and stored in disk. This was the case of 18.9% of the injected faults.

Despite the possibility that one of the remaining 68% software faults that have not been activated by the workload can lead to a different and not yet experienced failure mode occurring, this observation strongly suggests that the impact of the used toolstack is either non existent or very drastic and visible (hence relatively easy to handle). This generally means that the administrator should not need to worry about the impact that software faults can have when opting to change the default toolstack provided by Xen with another toolstack of his preference. This latter observation may in fact reduce the likelihood of

operator faults and contribute positively to the overall dependability of the system.

5.3 SOFTWARE FAULTS IN DEVICE DRIVERS

Faults injected in the device drivers aimed to investigate the effect that software faults in this component, which is shared by all VMs of a virtualized node, can have on the dependability of the node. A total of 3 353 out of all 9 986 possible faults (~33%) were injected with the outcomes as presented in Figure 20.

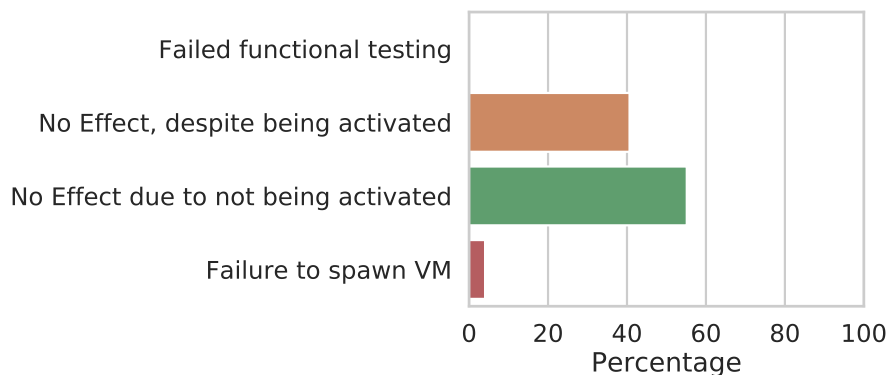


Figure 20: Failure modes and probabilities when injecting on a device driver.

Of these, 1 849 faults only affected non-executed lines and just 5 faults were detected by the functional tests, which consisted in starting a SSH connection. Of all faults, 4.1% lead to failures which caused the VMs to fail when spawning or to not be reachable (probably because the network connection failed). These failures are easily detectable due to abruptly crashing the program, returning error codes, displaying error messages, unreachable VMs, *etc.* and do not lead to silent data corruption.

5.4 RELATED WORK

Literature in the topic of software fault injection has dealt mostly with the representativeness and performance of the technique. A pivotal topic respects to the fault types (*i.e.*, what and how to inject) and fault locations (*i.e.*, where to inject) that should be used to ensure that the injected faults represent realistic software faults. Natella *et al.* [98] conducted millions of software fault injection campaigns over different applications and have shown that indiscriminately injecting faults in every location of a program yields a large amount of non-representative faults (*i.e*, faults that are detectable through testing and thus would not be present in real applications). The same paper studied with success the usage of supervised and unsupervised classification techniques that were trained with simple software metrics,

namely lines of code, McCabe complexity, fan in and fan out, as to comprehend their ability to classify the best locations of a software project to inject faults in order to maximize fault representativeness.

One of the most advanced and popular fault models for emulating software faults in C applications is derived from Durães *et al.* field study [42] and has been implemented by a range of fault injection tools, including G-SWIFT [42], SAFE [98], EDFI [54] and ucXception. The named field study analyzed 668 real software faults across 12 popular open-source applications and classified the reason (*e.g.*, missing initialization, error in branch condition) behind each fault. From this classification of real faults, a fault model containing the most common operators was created.

Earlier works in this topic have tried to emulate a software fault indirectly, such as by injecting data errors, interface errors and even hardware faults, however the representativeness of such fault models is debatable and some works suggest that injecting hardware faults [90] and faults at the interface-level [80, 93] does not accurately represent real software faults.

Although software fault injection has rarely been used to evaluate cloud computing or virtualized systems, it has been prominently used in the evaluation of operating systems, namely for studying how faults in device drivers [43, 65].

5.5 SUMMARY

Software faults are unavoidable in projects of reasonable dimension and even the most thorough software testing practices cannot ensure the absence of faults. Cloud computing is particularly affected by this problem due to its reliance on software for providing virtualization and managing its infrastructure. However, when it comes to virtualized and cloud computing systems, there is limited research analyzing in which components are software bugs more preponderant and how they affect these type of systems, namely when focusing on availability and reliability, as opposed to security. This chapter contributes to the state-of-the-art with an analysis of how two important components of a virtualized system fail due to software faults that are not detected during the software testing phase and reach production (*i.e.*, residual faults). To ensure that only residual software faults were injected in the fault injection campaigns, simple functional tests were developed and used to filter out the faults that would be detected during testing.

The main observations to take from the experimental evaluation are:

- Software faults in the toolstack caused solely failures to spawn VMs;
- Software faults in the device driver lead to failures to spawn VMs or to interact with spawned VMs;

- No occurrence of silent data corruption was detected.

These observations suggest that software faults when activated *cause mostly failures that lead the provided service to fail completely* but *do not cause data corruption*. A more detailed analysis of the results yield more specific observations, such as:

- The majority of faults affected non-executed lines, which reveals that even complex workloads are unable to exercise a large part of the code base, either because of exception handling routines, which are only entered in exceptional cases, or due to architecture-specific code (*e.g.*, x86 vs ARM);

- Simple hand-crafted functional tests were sufficient to filter out software faults that would likely have been detected before reaching production;

- The consistent and broad usage of *asserts* in the code base of xl/libxl lead to functional test being quite effective at detecting injected software faults, as unexpected conditions or corrupted state often lead to early errors being thrown;

- However, certain software faults lead to a stop in the middle of the process of spawning a VM, leaving it in a zombie-like state in which the VM is not executing but it still appears as up.

The nature of failures caused by software faults suggests that these failures can be easily detected and that relatively simple fault tolerance mechanisms can be applied. A technique such as N-version programming [28], despite its known drawbacks and limitations, should tolerate most failures due to software faults through data and design diversity. A possible application of N-version programming to a virtualized system could consist on the combination of two different toolstacks (*e.g.*, libxl and libvirt) or even two different hypervisors (*e.g.*, Xen and KVM).

Part III

IMPROVING CLOUD COMPUTING DEPENDABILITY

6 AVAILABILITY THROUGH MIGRATION TO A CO-LOCATED HYPERVISOR

Cloud computing is susceptible to outages that cause unavailability due to hardware, software and operator faults. A part of these faults may affect the hypervisor, which hosts multiple VMs over the same node, hence potentiating common-mode failures that affect more than one VM at once. The previous chapters examine the reasons behind these failures and provide a characterization of the failure modes that can occur, thereby supporting the key contribution of this chapter, Romulus, a fault tolerance technique for protecting VMs against hypervisor failures.

The goal of Romulus is to improve the availability of cloud computing deployments by maintaining service continuity of the VMs in the presence of hypervisor failure. Successfully resuming a VM after a hypervisor failure, *e.g.*, due to a transient hardware fault or a software fault, has not been considered so far in the literature because of the assumption that hypervisor failure necessarily leads to all VMs failing alongside, since the hypervisor has unrestricted access to the hardware, including every VM's memory space, and its failure may propagate to the VMs. However, we hypothesize that VMs are often left in a correct state after a hypervisor failure due to the results in Chapter 4 having shown state corruption to be a relatively uncommon occurrence (*e.g.*, about 2.5% of failures caused by soft errors during hypercall execution). In these situations, the hypervisor crashes or hangs, preventing any execution from taking place, and all VMs become unavailable (in spite of their internal state remaining correct). Fault injection experiments were conducted to show that this hypothesis holds, thus suggesting that VMs can be recovered after hypervisor failures.

Based upon this observation, we construct a generic and transparent fault tolerance technique to recover multiple VMs through migration between two hypervisors co-residing on the same physical hardware (more specifically, a failed hypervisor and a healthy hypervisor). A lightweight layer added between the hardware and the hypervisors monitors the state of the VMs, triggers recovery action on hypervisor failure and performs part of the VM migration process. This layer is an essential component that allows Romulus to provide fault tolerance without redundant hardware resources (*e.g.*, Remus [38] requires two physical hosts), in a transparent and generic manner (*i.e.*, without modifications to the VMs, operating system or applications) and with acceptable performance overhead.

A standalone proof-of-concept implementation was developed over Xen with a limited number of modifications. The source code has been made public as to foment development and interest from other members of academia and industry. The proof-of-concept was evaluated using fault injection, similarly to what has been performed in Chapters 4 and 5, in order to verify the validity of the hypothesis that sustains Romulus and the performance of the proof-of-concept. The results show that Romulus is capable of recovering at least one VM after the large majority of failures, whereas all VMs in the system can be recovered successfully but less often. Downtime is in the order of a few dozen seconds, which is accomplished by simply resuming VM execution instead of restarting it.

The source code of the Romulus proof-of-concept is available at `https://github.com/ucx-code/romulus`

6.1 APPROACH

Romulus performs migration of VMs from a failed and possibly uncooperative hypervisor to a co-located and healthy hypervisor as an approach for tolerating failures of the hypervisor, such as those caused by software faults and transient hardware faults. More specifically, Romulus handles failures of the hypervisor that cause it to hang or crash and, consequently, affect all the VMs running in the system, thus causing common-mode failures that lead to unavailability.

A novel aspect of Romulus resides in how its migration process is able to extract VM state from an unresponsive or uncooperative hypervisor in a transparent manner and then resume the VMs in another hypervisor. Minimizing migration duration is essential for increasing availability and Romulus accomplishes a low downtime, which is mostly spent performing state migration, because its VMs can resume operation immediately after migration and without requiring a costly reboot.

The success of Romulus depends on the error inside the hypervisor being isolated before it can propagate to the VMs and corrupt their state. As such, the fault detection mechanism plays an important role in this effort and in obtaining a balance between false positive and false negative rate. However, Romulus does not dictate the type or properties of the fault detection mechanism to be used, instead leaving this decision to the user.

6.1.1 *Architecture*

Romulus' architecture requires compliance with a set of requirements:

1. a microvisor, *i.e.* a minimal hypervisor, must be added directly above the hardware for managing the failure detection and migration process;

2. the microvisor must support nested virtualization;

3. above the microvisor, two full-fledged hypervisors must be instantiated;

4. the microvisor must support virtual machine introspection (VMI) to enable the migration of VMs between hypervisors;

5. the hardware platform must support hardware-assisted virtualization (*e.g.*, through Intel VT-x or AMD-V extensions);

6. the VMs to be recovered must be virtualized using hardware-assisted virtualization.

Figure 21 depicts the architecture of Romulus, including the three layers of virtualization that explain the need for the usage of nested virtualization [18]. The first layer (L0) is the microvisor, the second layer (L1) are the two hypervisors and the last layer (L2) are the VMs managed by the clients (IaaS) or the cloud provider-managed VMs providing a service (SaaS and PaaS).

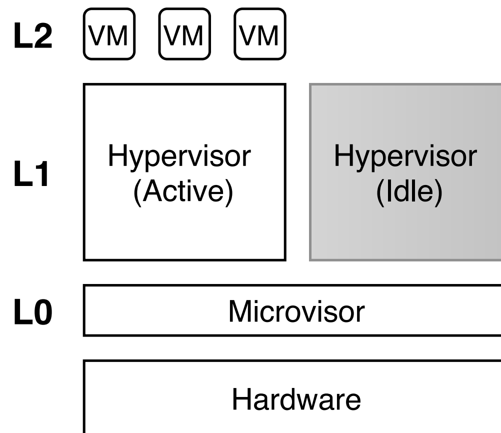


Figure 21: Architecture and layers of Romulus.

The microvisor is a barebones and lean hypervisor that implements only the most essential functionalities, such as virtualizing a VM, scheduling CPU execution and managing the system memory. When compared to a traditional hypervisor, the microvisor does not need to implement functionalities such as providing a complex interface for management by the user (*e.g.*, through a toolstack), containing device drivers to interact with the hardware, implementing mechanisms for inter-VM communication or memory sharing (*e.g.*, grant-based memory sharing and event channels), and more. In fact, the microvisor delegates as many functionalities as it can to other parts of the system and strives to occupy the least amount of CPU time that is possible, both in order to reduce its overhead on the system and to reduce its exposure against transient hardware faults. Moreover, its limited set of features means that the microvisor needs only a relatively small number of lines of code, which in conjunction with the usage of

classical software engineering techniques can lead to the creation of a robust piece of software.

This is important because the introduction of the microvisor replaces the single point of failure of a traditional virtualized system, which is the hypervisor and privileged VM, with the microvisor. Since the microvisor itself becomes the new point-of-failure, a reduced surface that is less susceptible to faults can contribute to a more resilient system.

Above the microvisor, two full-fledged hypervisors (such as Xen or KVM) are spawned. One of the hypervisors, which is named L1A, will be actively operating and hosting VMs, while the remaining hypervisor, known as L1B, will be idling or suspended without any VM running over it. This dual-hypervisor setup can be extended to include either a pool of idle hypervisors or a mechanism that destroys the active hypervisor after the migration process has been concluded and replenishes the system with a new idle hypervisor for future use.

Both hypervisors may either share the same implementation (*i.e.*, source code) or use different hypervisor implementations (*e.g.*, Xen and KVM). The first option is easier to configure and provides protection against transient hardware faults and certain software faults, such as mandelbugs and aging-related bugs, whereas the second option also provides coverage against other software faults (*e.g.*, some bohrbugs) through design diversity.

6.1.2 *Lifecycle*

The lifecycle of Romulus can be divided into three phases: preparation, monitoring and migration. The preparation phase is used to extract the static and semi-static state (*i.e.*, the state that does not change or rarely changes during a VM's lifecycle), specifically the configuration and emulation state, for each of the VMs that will be recovered when a hypervisor failure occurs. The monitoring phase takes up the majority of the lifetime and consists in two main actions: monitoring and storing VM state that may change dynamically, as is the case with some of the CPU state, and monitoring the health of the hypervisor and/or its VMs in order to detect when a recovery action must be performed. The exact approach used to monitor the hypervisor health and to trigger the recovery action when a failure is detected (*i.e.*, the triggering mechanism) is not the main focus of this article and any valid option may be combined with this failure recovery mechanism.

The first step in the migration phase is to pause the execution of the failed L1 hypervisor, in order to prevent corruption to its data structures or VMs. In the case that the hypervisor has already sent a shutdown signal (*e.g.*, as a response to a segmentation fault from hypervisor code), it is essential that the microvisor preserves its memory state (*i.e.*, it should not free the memory pages). Then, the

microvisor must employ virtual machine introspection to extract the missing VM state from the hypervisor's memory. This implies that the microvisor knows the offsets and sizes of the hypervisor's data structures and hence it requires a hypervisor-dependent configuration. When all of the required VM state has been obtained, the migration can take place. Firstly the memory of the VM is moved from the failed to the sane hypervisor. This operation is implemented by the microvisor without the need for memory copying, simply by rearranging its physical-to-machine table so that the memory pages that contain the state of the L2 VM and that were previously mapped by the failed L1 hypervisor now belong to the sane hypervisor (see Algorithm 1). Thus reducing the time required for migration.

Algorithm 1 Algorithm for memory state migration of one VM in an Intel system.

Input: EPTP
1: $pml4 = *EPTP$
2: change_ownership(pml4, L1A, L1B)
3: **for** $l0 = 0$ to 511 **do**
4: $pdpt = pml4[l0]$
5: **if** (pdpt is valid) **then**
6: change_ownership(pdpt, L1A, L1B)
7: **for** $l1 = 0$ to 511 **do**
8: $pd = pdpt[l1]$
9: **if** (pd is valid) **then**
10: change_ownership(pd, L1A, L1B)
11: **for** $l2 = 0$ to 511 **do**
12: $pt = pd[l2]$
13: **if** (pt is valid) **then**
14: change_ownership(pt, L1A, L1B)
15: **end if**
16: **end for**
17: **end if**
18: **end for**
19: **end if**
20: **end for**

In order for the microvisor to know which pages belong to a L2 VM, it must know the location in memory of the entry for the nested page table structure containing the physical-to-machine mapping that keeps the VM's pages, which is known as EPTP (EPT pointer) in Intel systems and nCR3 in AMD systems. This pointer represents an entry point to a multi-level paging structure which is used by every hardware-virtualized VM (in Xen this corresponds to the HVM virtualization mode) and which can be iterated to find all the pages of a VM.

After the microvisor exchanges the ownership of all of the memory pages from the failed hypervisor to the sane hypervisor, it should use the hypervisor's default suspension and resume mechanism for restoring the VM in the sane hypervisor. This step can be simplified by updating a template save file with the most recent known VM state (*e.g.*, configuration, CPU and emulation state) and using it for resuming the VM.

To complete the recovery of a VM, on the previously idle hypervisor's (now active) side, the capability for restoring a VM's memory state from a memory location containing the entry of the nested page table must be available. This restoration of a VM's memory differs from how most hypervisors' transfer memory state, which is by adding the memory page's contents to the save file and copying the contents into memory on VM restore. This method of restoring a VM's memory state is required in this situation due to the way the microvisor exchanges the ownership of memory between hypervisors (based on the entry pointer for the nested paging table), however it also brings a significant performance advantage by avoiding copying between memory and disk, which ultimately reduces the downtime during recovery.

After the recovery of all VMs is complete, the paused and failed hypervisor can be destroyed and its resources, including the memory pages that still belong to it, can be freed. At this point the lifecycle may restart, but not before a new idle hypervisor instance is spawned to serve as the sane hypervisor.

6.1.3 *Optimized state extraction*

One of the main elements of novelty in Romulus is the method for extracting VM state from a crashed or uncooperative hypervisor, which is an essential part of the process for tolerating and recovering VMs from a hypervisor failure. For this purpose, virtual machine introspection (VMI) [51], that is 'monitoring and analyzing the state of a virtual machine from the hypervisor level' [106], is performed by the microvisor, thus enabling extraction of the required state without hypervisor intervention as long as the internal structure is known *a priori*.

The state that is required for migration can be divided into:

1. *configuration state* – refers to details about the VM, such as how many memory pages, disks, CPUs, *etc.* it has;
2. *memory state* – refers to the memory pages assigned to a VM, which contain the data of the kernel and user space processes;
3. *CPU state* – refers to the register values for each of the CPUs used by the VM and to the data structures required by virtualization

extensions (*e.g.*, VMCS and VMCB which are mandatory when using Intel's VMX extension);

4. *disk image* – contains the information stored by the VM in its physical storage;

5. *emulation state* – refers to the auxiliary state needed by the hypervisor to perform the virtualization and usually refers to the state of I/O devices (disks, network interfaces, *etc.*).

Different state requires specific methods for obtaining, treating and reusing it. Configuration state tends to be static and defined at VM boot time, hence it can easily be pre-obtained. Memory state is obtained and restored by taking advantage of the available memory virtualization extension (Intel EPT, AMD NPT, *etc.*) which keeps a multi-level paging structure in memory that describes the memory structure of a VM. CPU state, in particular the register state, is obtained by introspecting the L1 hypervisor structures that keep track of each VM's CPU state. However, different hypervisor implementations may keep track of different amounts of state, and part of the state may not be stored by the L1 hypervisor. For these situations, the microvisor must keep track of the missing state itself by trapping nested exits from the L2 VM and storing the required state. Disk state can be obtained and shared between L1 hypervisors using a range of well-established techniques, such as through a network file system, hence we will not dwell on this point. Emulation state, while not very dynamic, may change (for example when a previously disabled network interface becomes active) and is particularly difficult to obtain, as this state is usually stored in user-space processes (*e.g.*, QEMU) and hence harder to find and obtain. However, it can still be obtained from the L1 hypervisor's memory using VM introspection, or simply by relying on an outdated, but possibly correct, snapshot that is obtained after start up of a VM.

6.2 PROOF-OF-CONCEPT

A proof-of-concept implementation of Romulus was created over the source code of the Xen hypervisor [14], which served as the basis for the microvisor, and reused the functionalities provided by libVMI v0.12.0, which was used with caching disabled, and Intel VT-x hardware extensions. If the proposed technique was to be used in a production environment, the microvisor would ideally be developed from scratch, with special attention given to reducing its code size, focusing on a target architecture, refraining from implementing non-essential functionalities and, possibly, using defensive programming or formal verification techniques.

6.2.1 *Modifications*

In order to adapt Xen 4.11.1 to serve as the microvisor that is required by Romulus, a total of ∼ 1K lines of code were modified that largely account for adding two new hypercalls. One such hypercall is denominated *save_nvmcs* and has the role of extracting and initiating the monitoring of the CPU state that is not tracked natively by Xen. More specifically, this hypercall activates the execution of a branch added to the *nvmx_n2_vmexit_handler* function (shown in Figure 22), which is called after the exit of a L2 VM, that stores the values related to the *es*, *fs*, *cs*, *gs*, *ds*, *ss*, *tr*, *gdtr*, *idtr* and *ldtr* segment registers. It should be noted that different procedures can be used to implement the step of CPU monitoring that is required by Romulus.

The other hypercall is called *migrate* and receives as input the EPTP of the L2 VM to be migrated and the memory location on the idle hypervisor to where the L2 VM's memory should be moved and performs the migration. The key part behind this hypercall is the *iterate_ept_structures* function (see Figure 23), which receives the address of an EPT structure, moves its ownership from one hypervisor to another using the *update_ept_for_new_host* function (see Figure 24), and iterates over the structure and recursively calls the next structure until reaching the last level of depth.

Furthermore, the toolstack was extended to support calling the aforementioned hypercalls and a range of supporting userspace utility applications was developed to: i) use VMI to obtain the EPTP and part of the CPU state of the L2 VM; ii) replace the contents of a base save file (created after the L2 VM has been spawned) with a more recent CPU state and the EPTP of the L2 VM on the idle hypervisor (after migration); iii) remove the page tables that Xen stores on the save file, in order to avoid higher migration time due to unnecessary information on the save file.

Only the code of the L1 B hypervisor needs modifications to support Romulus. These modifications provide the capability to restore a VM from a save file that contains an EPTP, thus extending over the normal functionality of Xen, which expects the memory content of the VM to be embedded on the save file. Part of these modifications led to the addition of a simple hypercall and matching toolstack code whose purpose is to prepare the hypervisor to receive the memory of the L2 VMs from the active hypervisor. This can be accomplished by calling the *alloc_domheap_pages* function to allocate the free memory and reserve it for future use.

6.2.2 *Lifecycle and flow*

Figure 25 presents the flow and actions taken during the lifecycle of a system that uses this proof-of-concept. The shown flow assumes

```
int nvmx_n2_vmexit_handler(struct cpu_user_regs *regs,
                           unsigned int exit_reason) {
  (...)
  struct vmcs_save_state * save_state;
  int slot_id;

  if (nvmcs_global_state.enabled == 1) {
    unsigned long sysenter;
    __vmread(GUEST_SYSENTER_EIP, &sysenter);

    if (sysenter != 0) {
      spin_lock(&nvmcs_lock);
      slot_id = nvmcs_has_eptp(sysenter);
      if (!((slot_id == -1) && (nvmcs_global_state.next_is_domid ==
          0)) {
        if (slot_id != -1) {
          save_state = &nvmcs_global_state.save_states[slot_id];
        } else {
          save_state = &nvmcs_global_state.save_states[
              nvmcs_global_state.free_slot];
          save_state->eptp = sysenter;
          save_state->domid = nvmcs_global_state.next_is_domid;
          nvmcs_global_state.next_is_domid = 0;
          ++nvmcs_global_state.free_slot;
        }

        __vmread(GUEST_SYSENTER_CS, &save_state->sysenter_cs);
        __vmread(GUEST_SYSENTER_ESP, &save_state->sysenter_esp);
        __vmread(GUEST_ES_SELECTOR, &save_state->es_sel);
        (...)
        save_state->ready = 1;
        wmb();
      }
      spin_unlock(&nvmcs_lock);
    }
  }
  (...)
}
```

Figure 22: Code added to nvmx_n2_vmexit_handler.

```
static int iterate_ept_structures(struct domain * d,
        unsigned long addr, int level, int page_order,
        struct domain * t, ept_entry_t * parent_epte) {
  ept_entry_t * mapping;
  unsigned long mfn, new_gpfn, new_tmp; int i;
  struct page_info * page;

  page = get_page_from_gfn(d, addr, NULL, P2M_ALLOC);
  mfn = mfn_x(page_to_mfn(page));
  new_gpfn = update_ept_for_new_host(d->domain_id, t->domain_id,
      addr, page_order, level, parent_epte, page, mfn);

  if ((page_order != PAGE_ORDER_4K) || (level == 4)) {
    return new_gpfn;
  }

  mapping = (ept_entry_t *) map_domain_page(_mfn(mfn));
  for (i = 0; i < 512; i++) {
    ept_entry_t tmp = atomic_read_ept_entry(&mapping[i]);
    if ((is_epte_valid(&tmp)) && (is_epte_present(&tmp))) {
      if (is_epte_superpage(&tmp)) {
        int page_order2 = (level == 1) ? PAGE_ORDER_1G :
            PAGE_ORDER_2M;
        new_tmp = iterate_ept_structures(d, (unsigned long) tmp.mfn,
            level + 1, page_order2, t, &tmp);
        &mapping[i]>mfn = new_tmp;
      } else {
        new_tmp = iterate_ept_structures(d, (unsigned long) tmp.mfn,
            level + 1, PAGE_ORDER_4K, t, &tmp);
        &mapping[i]>mfn = new_tmp;
      }
    }
  }

  unmap_domain_page((void *) mapping);
  put_page(page);
  return new_gpfn;
}
```

Figure 23: Code of the iterate_ept_structures function that was added to Xen.

```
1  static int update_ept_for_new_host(unsigned int source_domid,
         unsigned int target_domid, unsigned long source_gpfn,
         int page_order, int level, ept_entry_t * parent_epte,
         struct page_info * page, unsigned long mfn) {
     unsigned long target_gpfn;
6    struct domain * sourceD, * targetD;

     sourceD = rcu_lock_domain_by_id(source_domid);
     targetD = rcu_lock_domain_by_id(target_domid);

11   target_gpfn = find_next_free_gpfn(targetD, page_order);

     spin_lock(&sourceD->page_alloc_lock);
     page_list_del(page, &sourceD->page_list);
     spin_unlock(&sourceD->page_alloc_lock);
16
     guest_physmap_remove_page(sourceD, _gfn(source_gpfn), _mfn(mfn),
          page_order);
     guest_physmap_add_entry(targetD, _gfn(target_gpfn), _mfn(mfn),
         page_order, p2m_ram_rw);

     spin_lock(&targetD->page_alloc_lock);
21   page_list_add_tail(page, &targetD->page_list);
     page_set_owner(page, targetD);
     spin_unlock(&targetD->page_alloc_lock);

     rcu_unlock_domain(sourceD);
26   rcu_unlock_domain(targetD);

     return target_gpfn;
   }
```

Figure 24: Code of the update_ept_for_new_host function that was added to Xen.

that, at the start, all L1 and L2 VMs are up and running and that the save files and I/O state of the VMs is accessible to the microvisor and both L1 hypervisors. A total of nine actions have been identified and grouped into the three phases that constitute the lifecycle of Romulus.

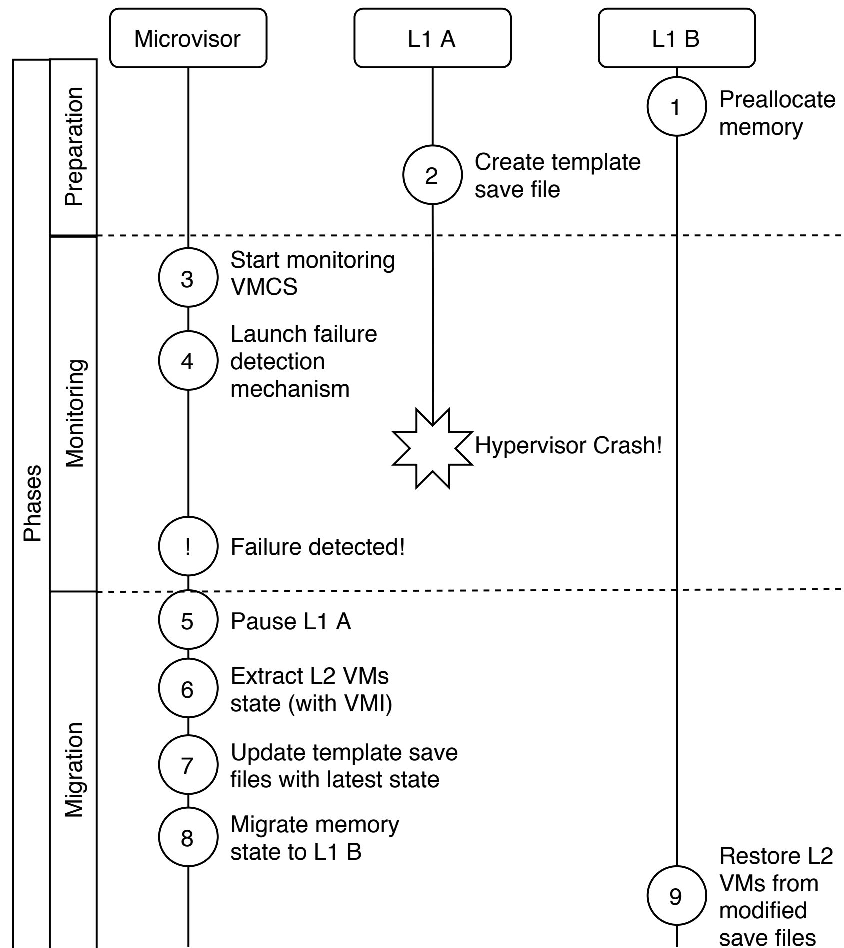


Figure 25: Lifecycle and actions.

6.2.3 *Limitations*

Considering that the presented implementation is a proof-of-concept, it contains several limitations of its own that are not inherent to Romulus. One limitation of this implementation is that recovery is only supported for L2 VMs that have just a single vCPU, do not use hyperpaging (*i.e.*, page sizes bigger than 4 Kb), use an Intel EPT hardware virtualization mode (HVM in Xen), do not use Xen's PV-on-HVM drivers (*e.g.*, by passing 'nopv' parameter to recent Linux kernels). Furthermore, LAPIC, APIC, MCE, XSAVE and X2APIC should not be used in the L2 VMs and hyperpaging must be disabled on both L1 hypervisors, but may be enabled in the microvisor.

6.3 METHODOLOGY

To verify the effectiveness of Romulus in fulfilling its objectives, namely providing fault tolerance against transient hardware and software faults that affect the hypervisor, an experimental setup and methodology was designed following many of the approaches and techniques that have been used in past experiments. For example, fault injection was once again employed to emulate the effect of hardware and software faults. On the other hand, a new workload was developed and used for the first time in this chapter, as the experience acquired throughout the development of this thesis has lead to improvements over the workloads that have been used previously.

6.3.1 *Physical setup*

The setup supporting the experiments is comprised of two different physical systems, as depicted in Figure 26. One of the systems has the role of 'Compute Node' (*i.e.*, hosting the VMs) and the other is used as the orchestrator, client and disk image provider for the VMs. The compute node is equipped with two Intel Xeon Silver 4114, each with ten physical cores, 32 Gb of RAM and a network interface capable of 1 GbE. The orchestrator machine is equipped with a single Intel Xeon E5620 with four physical cores, 12 Gb of RAM and a 1 GbE network interface.

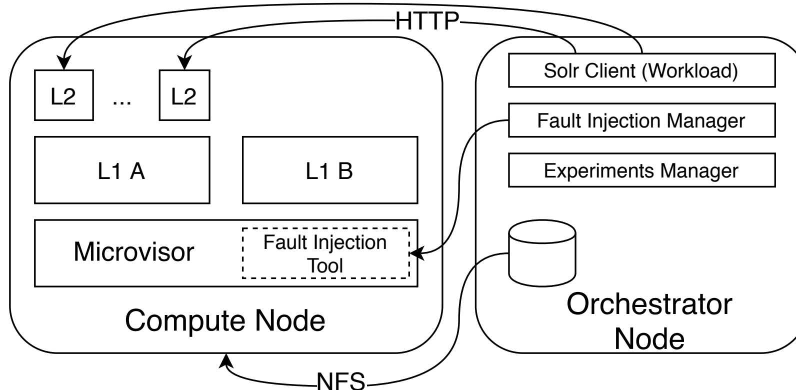


Figure 26: Experimental setup used for the experiments.

The disk images used by the VMs that are running on the compute node are stored and provided by the orchestrator machine through NFS [129]. This is a common setup found in the cloud (although using more advanced technologies [47]) where there is one or more nodes dedicated to storing and providing disk resources.

In the experiments where fault injection was used to emulate transient hardware faults both hypervisors used Xen 4.11.1, whereas when software faults were injected L1A used Xen 4.12.3 and L1B used Xen 4.11.1. The usage of different hypervisor versions reflects a scenario

where Romulus is used to tolerate software faults that cause transient failures (*e.g.*, mandelbugs and aging-related bugs) through state rejuvenation and software faults that cause permanent failures (*e.g.*, bohrbugs) through design diversity. If the same hypervisor version had been used instead, Romulus would only have been able to cover software faults of transient nature.

6.3.2 *Workload and profiles*

A new workload was built and used in the experiments described in this chapter. This workload emulates a Solr server [132] that provides access to a part (11 Gb) of Wikipedia's index [153]. It is a CPU and IO-heavy workload that also exercises memory and represents one of the various workloads often found in cloud deployments. Only search operations are performed during the course of the workload, in order to avoid having to keep track of state changes in the workload client, which would complicate the task of verifying the impact of the injected faults.

Along with the definition of the workload itself, various profiles, which configure the various parameters of the workload, were created. More precisely, a total of two different profiles were used: a *full load* profile, which consists in 25 clients performing a request once every 50 milliseconds and represents a worst-case scenario where the VM is overloaded, and a *light load* profile, which represents a more common scenario [15, 25, 39, 86, 114] where the system seldom reaches full resource usage and consists of one client performing a request every second. Figure 27 and Table 16 provide insight into the resource usage of both workload profiles on a VM with 3 000 Mb of RAM.

6.3.3 *Fault injection*

Fault injection was employed to emulate realistic failures of the hypervisor, namely those failures caused by transient hardware faults in CPU registers and software faults in hypervisor code. To emulate transient hardware faults, a fault injection tool of the ucXception framework (namely, the tool described in Section 3.2.2.2) was used along with a single bit-flip fault model [121] targeting CPU registers.

For injecting software faults in hypervisor code, another tool of the ucXception framework was used (more precisely, the tool described in Section 3.2.2.3). This tool generates patch files according to the fault model, which are then applied to the source code of the L1A hypervisor, one patch per each run during the fault injection campaigns. In order to speed up the evaluation, an analysis of the lines of the hypervisor source code that are covered during the workload was performed and all faults that do not affect those lines were filtered out.

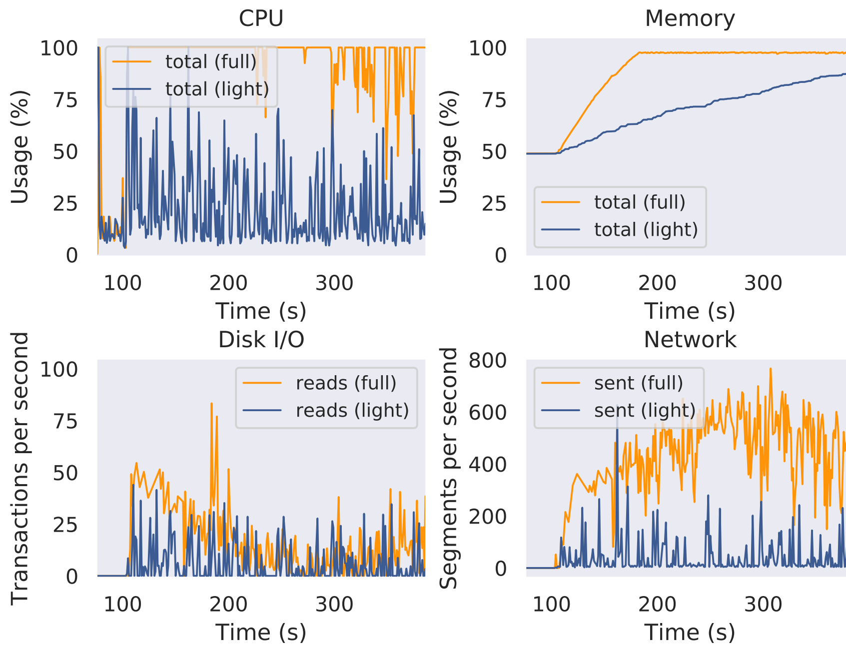


Figure 27: Resource usage of the workload and its profiles.

		FULL LOAD			LIGHT LOAD		
		MAX	AVG	MED	MAX	AVG	MED
CPU	Total (%)	100	88	100	100	21	14
	User (%)	83	30	30	89	9	6
	Kernel (%)	90	57	66	68	8	4
	IO Wait (%)	28	0.3	0	48	4.3	0
Disk	Read (TPS)	99	13.6	9.6	104	7	0
	Read (Mbps)	65	9.7	5.6	73.8	5	0
	Write (TPS)	68	1.3	0	38.6	1	0
	Write (Kbps)	415	20.5	0	280	12	0
Network	Sent (seg./s)	766	365	423	625	33	8
	Recv. (seg./s)	682	360	420	376	30	12

Table 16: Statistical analysis of both workload profiles.

A requirement we set for our experiments was that the software fault should only be activated while the workload (*i.e.*, Solr) is being executed, thus after any preparatory steps (*e.g.*, creating base save files) have been completed. This was reached by encompassing the software fault inside an *if* whose activation depends on a variable (a similar practice has been used in other works [36, 161]) that is manipulated by the microvisor using VMI. When the value of the variable is false, the non-faulty code is executed, whereas if the variable is true, the faulty code is executed. Furthermore, for tracking purposes the exact moment when the fault is first activated is stored, as well as the number of iterations that occurred before and after the fault was activated.

Faults were only injected in lines of the L1A hypervisor that do not exist in L1B hypervisor, since L1A uses Xen 4.12.3 and L1B uses Xen 4.11.1, as to emulate recovery from a software fault that was introduced in a more recent version of Xen and that does not exist in older versions. A comparison between the source code of both versions was made which showed that about 8% of lines of source code (comments were disregarded) differed between the two versions, including lines of source code in files that implement essential functionality, thus suggesting that migrating VMs between different versions of the same hypervisor, even if both versions are relatively recent, can provide some coverage against software faults through design diversity.

When performing fault injection of hardware faults, twelve registers were targeted (*rip*, *rsp*, *rbp*, *rax*, *rbx*, *rcx*, *rdx*, *r8-15*) which represent all the registers that can be targeted using this fault injection tool. For software fault injection, the fault model includes 10 operators out of the 13 operators defined in Durães *et al.* fault model (see Table 3) and uses 2 operators belonging to the extended fault model [11] (namely, WLEC and WALR). In summary, the used operators were WVAV, WPFV, WLEC, WAEP, MVAV, MVAE, MLAC, MIFS, MIEB, MIA, MFC and WALR. Software faults were injected in 4 different source files, namely `arch/x86/hvm/vmx/vmx.c`, `arch/x86/hvm/vmx/vmcs.c`, `arch/x86/msr.c` and `arch/x86/mm.c`, which were chosen due to being the files that had the highest amount of changed lines between Xen 4.11.1 and Xen 4.12.3 that were exercised by the workload. These files contain functionality that deals with memory virtualization, hardware-assisted virtualization and model-specific register (MSR) emulation.

6.3.4 *Triggering mechanism for the recovery action*

Romulus depends on an error detection mechanism capable of understanding when a hypervisor has failed as to commence the recovery process. Since Romulus does not prescribe a specific error detection mechanism, leaving its choice up to the user, multiple options were considered for this role in our experiments. The goal in selecting the

best alternative for the recovery mechanism resides in reducing the downtime incurred from the detection interval and increasing the precision and sensitivity of the mechanism (*i.e.*, reducing the false positives and false negatives). For our experiments we opted to use a straightforward mechanism which consists in a constant stream of ping requests which will trigger recovery after a certain amount of consecutive pings has timed out. This proved sufficient for our requirements, which were to detect occasions where the hypervisor had a crash or hang failure which caused all the other VMs to fail (common-mode failure), however it might not be capable of detecting more complex failures where silent data corruption has occurred. Furthermore, it represents a baseline error mechanism, which means that better error mechanisms may possibly result in even better performance of Romulus than what is presented in this chapter.

6.3.5 *Correctness verification*

The workload client is designed to verify the correctness of the received responses depending on the query that was searched. This is accomplished by having a pre-prepared list of all possible search queries and the expected response, and then comparing all performed queries and respective response against the oracle, during workload execution. This step is essential in ensuring that failure recovery can be accomplished without corrupting the VM state in a manner that leads to incorrect output being sent to the service's clients.

6.3.6 *Recovery assessment*

Apart from ensuring the correctness of the responses, there is the need to assess whether each VM recovered successfully from a failure of the hypervisor. For this purpose two different 'points-of-view' are considered:

1. *Service point-of-view* – Recovery is measured according to whether the service (Solr) continued to execute correctly after the hypervisor failure. This is the traditional point-of-view, which corresponds to how the service clients experience unavailability;

2. *Operating system point-of-view* – A VM is considered recovered if its operating system continues to operate correctly, even if the service stops working. The state of the operating system is verified by performing a SSH check and executing a small number of simple commands at the end of each experiment. We consider this point-of-view because, although the service may not be successfully recovered, the operating system may continue to work correctly and specific methods can be designed to take advantage of this.

6.4 EVALUATION

This section presents the results obtained from the experiments that were performed, namely the experiments to verify what happens to the VMs when the hypervisor fails, which served as the motivation for developing Romulus, and the experiments that aim to evaluate the proof-of-concept implementation.

6.4.1 *What happens to VMs when the hypervisor fails?*

When a hypervisor fails, it may fail without affecting the state of its VMs or it may gain an erratic behaviour and eventually corrupt its VMs. If recovery is to be attained the basic principle that the VM state has to remain sane must be upheld. To evaluate this hypothesis, upon which Romulus depends on, an experimental campaign was setup where a single VM was migrated to a new hypervisor after its hypervisor failed, as to assess whether it continues correct execution.

Fault injection in CPU registers during hypervisor execution between 15 and 40 seconds after the start of the workload was used to generate hypervisor failures. The timing values were picked as to allow the VM to warmup while providing enough time for it to recover and resume responding to Solr requests before the workload finishes. Both profiles of the workload (light and full load) were evaluated during 200 seconds. The recovery process is triggered by a process that constantly performs ping checks to a VM and starts the recovery process sensibly 8 seconds after the last successful response.

A total of 339 failures using the full load profile and 663 failures using the light load profile were obtained, the results of which are displayed in Table 17. Over the course of all experiments, a total of 55 678 requests were performed and no occurrence of incorrect response data was detected after a successful VM migration to a new hypervisor. Although this does not eliminate the possibility that a VM may be successfully recovered despite its state being partially corrupt due to the fault in the hypervisor having propagated, which may then lead to silent data corruption occurring inside the VM, such is not a common occurrence. Moreover, the lower the detection interval before triggering recovery action, the higher is the chance of preventing errors in the hypervisor from propagating to the VMs.

A subset of all the runs, amounting to 102 failures using the light load and 155 using the full load profile, was extended to include a operating system check through SSH at the end of the run, in order to detect situations where the VM may be responsive but the Solr process has failed. The results indicate that in many situations, specially when a heavy workload is used, the operating system of the VM is able to recover but the application under use (Solr) does not, as it is killed by

the operating system. There was no run where Solr was classified as responsive but the operating system was unresponsive.

		Workload Profile	
		Light Load	Full Load
Point-of-View	Solr	189 (29%)	54 (16%)
	O.S.	40 (39%)	64 (41%)

Table 17: Recovery probability (1 VM).

The light load profile resulted in a higher recovery percentage than when using the full load profile, which suggests that the usage level of a VM can affect its likelihood of recovery. This is not a significant problem for cloud computing systems since the majority of the dozens of VMs consolidated on a physical system are idle or have a small amount of load during short periods of time [25, 39, 86] and the presented results can be considered to be a worst-case scenario for this setting, specially when referring to the full load profile.

6.4.2 *Can Romulus provide tolerance against hypervisor failures?*

Romulus' objective is to tolerate common-mode hypervisor failures that affect multiple VMs at once, hence the most adequate metric to evaluate the effectiveness of the proof-of-concept implementation is the total number of recovered VMs and, indirectly, the probability of recovery of a VM.

For this study a system containing four VMs with one CPU and 900 Mb of memory each and executing the Solr workload with the light load profile was used. Fault injection of transient hardware faults in CPU registers and software faults in hypervisor code was once again used to produce realistic failure data. Injection took place between 200 and 210 seconds after the start of the workload and the recovery action was triggered sensibly 40 seconds after the last successful ping reply. This is a conservative timeout value as to avoid inadvertent triggering, but which may reduce the recovery effectiveness by providing the failed hypervisor more time to corrupt the VMs state. The L1 hypervisors were configured to have six CPUs, which means that the system had a consolidation ratio of 0.66 (or in other words, four L2 VMs executing over six CPUs).

A total of 774 hypervisor failures due to injected hardware faults and 117 failures due to software faults were collected. To obtain these failures, we had to inject over 2 000 hardware faults and over 400 software faults. A part of these faults never propagated into a failure and thus have been excluded from our analysis. Table 18a shows information about the registers that lead to failures, while Table 18b

presents information about the operators that caused software failures. It should be noted that some CPU registers and operators did not cause any failures and have been omitted from these tables, nevertheless fault injection using these registers and operators was performed as usual.

REGISTER	FAILURES	
rip	141	(18%)
rsp	144	(19%)
rbp	68	(9%)
rbx	89	(11%)
rcx	85	(11%)
rdx	100	(13%)
r12	67	(9%)
r13	80	(10%)
Total	774	

(a) HW faults.

OPERATOR	FAILURES	
MFC	31	(26%)
MIA	49	(42%)
MIEB	1	(1%)
MIFS	21	(18%)
MLAC	1	(1%)
MVAE	2	(2%)
MVAV	2	(2%)
WAEP	2	(2%)
WLEC	3	(2%)
WPFV	3	(2%)
WVAV	2	(2%)
Total	117	

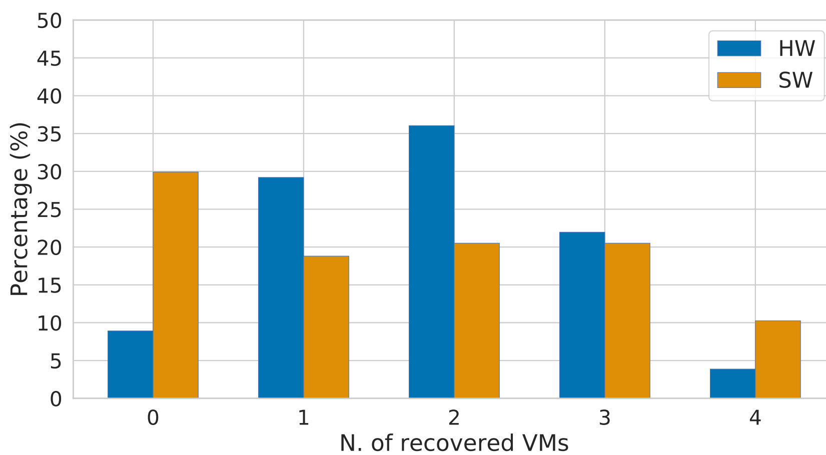
(b) SW faults.

Table 18: Fault injection statistics.

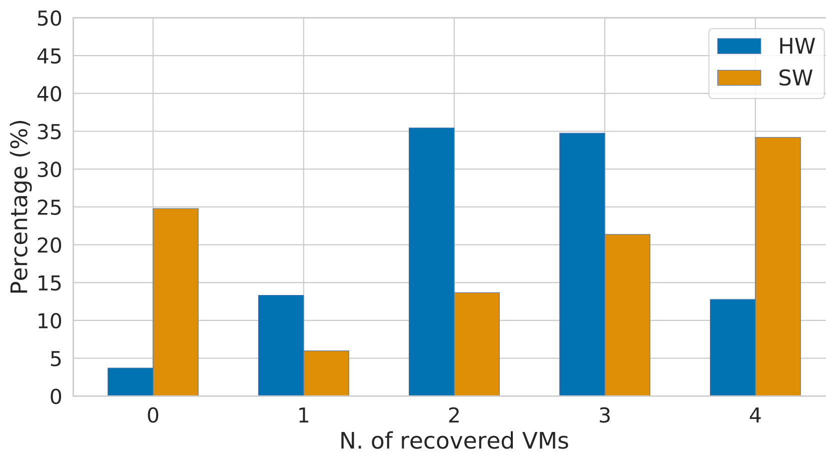
Figure 28 shows the histogram with the count of successfully recovered VMs discriminated by the type of fault, for the service and operating system point-of-view.

Hypervisor failures caused by transient hardware faults translated to at least one VM being successfully recovered (between 96% and 91% of all failures) and an arithmetic mean of 1.82 recovered VMs (or 46% of all VMs). When a hypervisor fails due to a software fault, the mean becomes slightly lower at 1.62 recovered VMs (41% of all VMs), but the extremes increase (no VM is recoverable between 25% and 30% of the time, whereas all four VMs are recovered between 10% and 34% of the time). These observations suggest that transient hardware faults, due to their nature, are more likely to propagate to a smaller number of VMs, leaving the remaining VMs in a correct state and thus recoverable, whereas software faults are more likely to affect and corrupt either all VMs or none at all.

Comparing both of the point-of-views that were considered for classification of a recovery outcome, the service point-of-view, which is the most restrictive of the two, experiences lower recovery probability than the operating system point-of-view. Situations where the operating system was left operational but Solr was not are explained



(a) From the service (Solr) point-of-view.



(b) From the operating system point-of-view.

Figure 28: Total recovered VMs after a hypervisor failure.

by the hypervisor failure having corrupted only state associated with Solr, which lead the operating system to kill its process after recovery. In these situations the addition of a simple application restart mechanism could greatly increase the recovery likelihood, although such would lose the transparency inherent to this fault tolerance technique. Figure 29 provides cumulative histograms that improve the comprehension of the data in the previous figures.

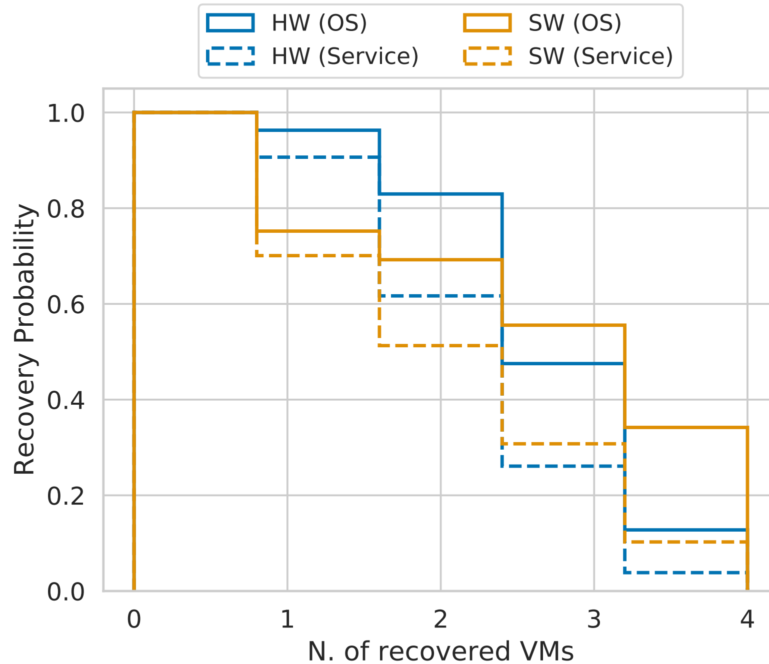


Figure 29: Cumulative histogram of recovery probability.

The operating system of all four VMs is recovered in 34% of all failures caused by software faults, but Solr is only recovered 10% of the time. The probability of all four VMs being recovered after a failure due to hardware faults is much lower, ranging between 13% for recovery of only the operating system and 4% for recovery of Solr. The recovery percentage increases if we consider all cases where at least three VMs were recovered: 56% of failures caused by software faults leave the operating system recoverable, but Solr can only be recovered 31% of the time, whereas if a failure is caused by a hardware fault, the operating system is recoverable 48% of the time and Solr 26% of the time. If we consider the operating system point-of-view, at least half of the VMs can be recovered in 83% and 69% of all failures, when caused by hardware and software faults respectively, or 62% and 51% of the time, if we use the service point-of-view. Finally, at least one VM is fully recoverable in 91% of all failures due to hardware faults and 75% of all failures due to software faults. Ultimately, even the recovery of one VM corresponds to a big improvement in comparison to a system that cannot tolerate hypervisor failures.

6.4.3 *How much downtime is incurred during the migration process?*

The downtime associated with the migration process has an important role on the overall availability of the system and must be evaluated. Figure 30 shows a boxplot depicting the median and quartiles of the downtime for a single VM setup using both the full and light load profiles, as perceived through the clients of the Solr service (service downtime) and through the operating system logs (VM downtime) obtained using the SAR utility.

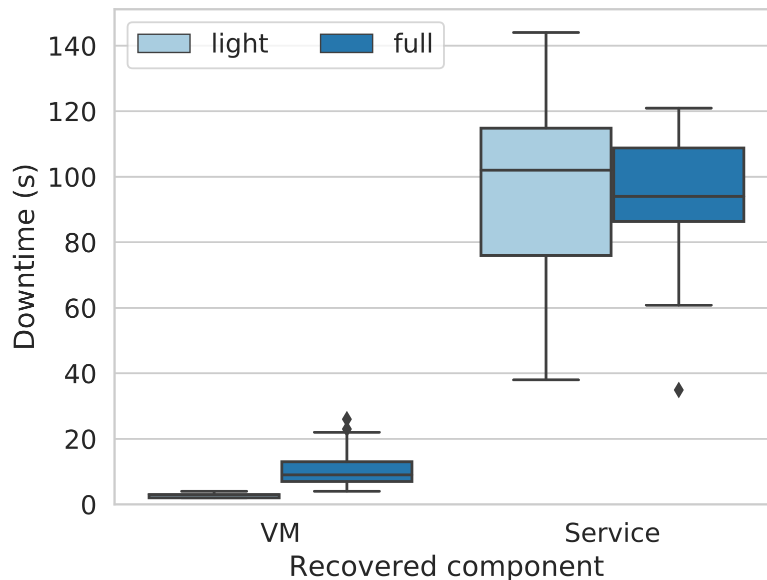


Figure 30: Downtime in a single VM setup.

The VM downtime reflects the period when the VM is operating but the network connection has not yet been restored, hence being considerably lower than the service downtime, ranging between 2 and 26 seconds with a mean of 3 seconds. The discrepancy between VM and service downtime is explained by a timeout in the network device driver used in the Linux kernel of the VMs that is only triggered after some seconds. Modifications to this device driver, or the usage of another driver, would equalize the VM and service downtimes.

A more detailed analysis shows that the large majority of time spent performing migration is due to memory page migration. Since the proof-of-concept implementation does not support hyperpaging in the L2 VMs, it suffers a lot of overhead in this step that could be avoided otherwise. Figure 31 shows an analysis of the contributors to migration time in an one VM system, using the light load profile and with memory sizes varying from 1 000 Mb to 8 000 Mb inclusive, at steps of 1 000 Mb. Each step represents the average of 30 runs.

The line with circle markers, labeled as *VM state migration*, represents all of the steps required for recovery apart from restoring the

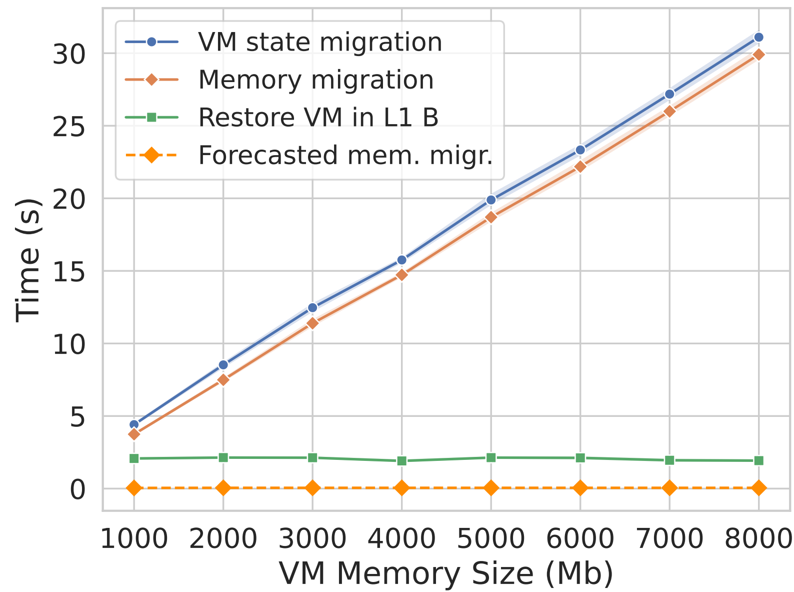


Figure 31: Duration of the recovery process per VM size.

VM state at the L1B hypervisor, which is represented by the line with square markers, while the solid line with diamond markers, labeled as *Memory migration*, describes the time taken solely for the step of migrating the memory state. The total time required to complete the recovery process is not depicted but consists on the sum of the lines that have circle and square markers.

While the time taken by the recovery process increases linearly with the memory size, and hence with the amount of migrated pages, the time to restore a VM remains somewhat constant independently of the VM memory size. The dashed line represents the hypothesized time that the step of memory migration would have taken if the proof-of-concept implementation had support for hyperpaging. It was obtained by measuring how many pages would be used at the different memory levels when hyperpaging is enabled and extrapolating from the known data. This extrapolation transmits a very positive outlook for the improvements that hyperpaging can bring, since it significantly lowers the amount of pages that need to be migrated, thus reducing time spent migrating the memory state to values between 45 and 54 milliseconds. According to this data, if hyperpaging is considered we can expect the entire recovery process to take between 2 and 4 seconds.

6.4.4 *How much performance overhead does Romulus introduce?*

As with any fault tolerance technique, overhead may be introduced in the system. We compared the performance, measured as the number of successfully answered requests over a fixed time period and their

average response time, of a system that represents a traditional virtualized cloud computing system (*i.e.*, where no nested virtualization is used), a system that uses nested virtualization but does not contain the proposed fault tolerance technique and a system with our implementation of Romulus. A total of 30 runs were performed for each setup and all the runs used a single VM with 3 000 Mb of RAM and the full load profile during 60 seconds. The L0 and L1 hypervisors, when applicable, were configured with 1 CPU and 12 000 Mb of RAM, as to represent a situation where consolidation is present. Table 19 presents the results for the light load and full load profiles, including a comparison of relative performance difference against the traditional virtualized system, which show that the proof-of-concept does not bring a measurable overhead on a system that already uses nested virtualization, however the addition of nested virtualization carries a significant overhead, which can reach up to 2.5x lower performance when a heavy workload is used.

		TRAD.	NESTED VIRT.	W/ POC
Full	N. of requests	1234	487 (+153%)	488 (+153%)
	Avg. response time (s)	1.18	3.14 (-62%)	3.13 (-62%)
Light	N. of requests	96	79 (+22%)	79 (+22%)
	Avg. response time (s)	0.09	0.24 (-62%)	0.24 (-62%)

Table 19: Performance overhead of Romulus compared against different setups.

6.5 LIMITATIONS

Beyond the limitations of the proof-of-concept implementation, which have already been discussed, Romulus, the fault tolerance technique itself, also has its own limitations. The first and foremost limitation of Romulus is its dependency on hardware-assisted virtualization, which means that it cannot be applied to VMs that are virtualized using other virtualization modes (*e.g.*, paravirtualization). Another limitation is the usage of nested virtualization. Nested virtualization is a relatively recent technique that has had significant adoption in the last few years, however it adds an additional layer of performance overhead to a virtualized system, despite the best efforts from developers of nested virtualization to reduce this overhead. As has been shown, nested virtualization constitutes the primary factor that introduces performance overhead to Romulus. Nevertheless, we expect that future developments regarding nested virtualization will also carry a positive effect to Romulus.

With regards to the experimental evaluation described in this thesis, some limitations should be referred. One limitation is due to only one workload having been evaluated, which is insufficient to cover the wide range of workloads that are commonly used in cloud computing deployments. Another limitation is due to the relatively small number of VMs that were hosted in the same physical system (more precisely, 4 VMs). The number of VMs was chosen while taking into account a trade-off between the time taken by each individual run and a representative amount of consolidated VMs. These limitations are not expected to invalidate the presented results, but future work will try to comprehend how different setups affect the performance of the technique.

6.6 RELATED WORK

High-availability operation has traditionally implied expensive setups that require redundant nodes that may be geographically far away [3, 44]. Despite the significant associated cost and performance overhead, the most common mechanisms for ensuring availability in cloud computing continue to follow such a distributed approach. For example, Remus [38] relies on a secondary passive host that constantly receives updates with the most recent state of the VMs and which takes control when a failure is detected in the active host. Hence it is capable of tolerating transient and permanent hardware faults that cause a full-stop crash of the system without corruption of the VMs, at the expense of twice as much hardware resources and a performance hit due to the action of snapshotting a VM and sending its state over the network.

In the same fashion, COLO [41] performs VM replication using an active-active and on-demand approach that monitors the external output produced by the VMs to trigger replication. As such, a VM only needs to be replicated when its output can be detected to differ, by comparison between both replicas. While PLOVER [149] combines active-passive backup with state machine replication across three nodes, thus overcoming some limitations of Remus and avoiding expensive state transfer.

Other available fault tolerance mechanisms for virtualized and cloud computing systems tend to rely on resetting the state of the applications, VMs or hypervisor to a known correct state, in order to eliminate errors in that same state, usually through microrebooting [22] (*i.e.*, rebooting of fine-grained components in order to renew possibly corrupted state and hence recover from a failure). Such is the case of ReHype [83], whose approach applies microreboot to the hypervisor as to provide fault tolerance against hypervisor failures. Its approach temporarily pauses the VMs while the hypervisor is rebooted and specific data structures are renewed with safe values. Results, obtained from a fault injection campaign of single bit-flips into registers during

execution of the hypervisor, show decent VM recovery performance (over 90% probability of recovery of at least one VM in a three VM system and around 70% chance of recovery of all three VMs) with no performance overhead. A posterior work [84] states a basic recovery latency of less than 3 seconds for a single VM, measured through ping timings, which is then reduced down to around 700 ms. A derivation of ReHype that is based on microreset is able to attain almost as good recovery rates with a recovery downtime close to 20 ms [159].

Similar fault tolerance techniques are RootHammer and HyperFresh. RootHammer [78] aims to reduce the time required for a normal hypervisor reboot by maintaining the VM state in memory during this process and quickly resuming the VMs after the reboot has completed. HyperFresh [9] presents a technique based on nested virtualization [18] and memory co-mapping for replacing a possibly corrupt and unstable hypervisor with a fresh hypervisor in as low as 100 ms, which can be employed as a software rejuvenation [68] mechanism to recover from transient software failures caused by latent and non-deterministic software faults.

Nested virtualization is commonly used to support various fault tolerance mechanisms for virtualized systems, including the already mentioned HyperFresh, as well as DualVisor and TinyChecker. DualVisor [155] uses redundant VM execution and data structures on the same physical host as a technique for detecting and tolerating errors caused by hardware faults. TinyChecker [140] uses nested virtualization to support monitoring of the communication between VMs and hypervisor and to duplicate key data structures, in an effort to detect and protect VMs from a misbehaving hypervisor.

6.7 SUMMARY

Hypervisor failures are a threat to the availability of cloud computing systems because they may propagate to the dozens of VMs that are usually consolidated on the same physical hardware and cause service disruption. Thus far the assumption was that whenever the hypervisor failed, all VMs were lost. On the contrary, we empirically show that a significant percentage of VMs remain correct after hypervisor failure and could be resumed in another hypervisor.

Based on this observation, we developed Romulus, a technique for tolerating hypervisor failures without requiring spare redundant hardware nor modifications to the virtual machines, which means that legacy applications executing in the cloud are natively supported. It performs efficient migration of the VM state from the failed hypervisor to a co-located hypervisor with the purpose of continuing VM execution after failure.

A proof-of-concept implementation of Romulus was developed over Xen, solving various technical challenges regarding tracking VM state

and performing VM migration across co-located hypervisors, which has been made available using an open-source license.

An experimental evaluation was conducted using fault injection of transient hardware faults and software faults, which concluded that nearly half of the VMs in a virtualized system are left in a non-corrupted state after a hypervisor failure and thus can be recovered if the failed hypervisor is replaced by a working one. The evaluation also showed that the proof-of-concept is capable of recovering an average of 41-46% of the VMs in the system after a hypervisor failure, while incurring a VM downtime in the range of a few seconds, which accounts in large part for the time taken to migrate the VM state between hypervisors. Despite occasions where not all VMs were successfully recovered, at least one VM was recoverable in the large majority ($\geqslant$ 75%) of cases, which, in itself, represents a big improvement over a traditional virtualized system where hypervisor failure always translates to the loss of every VM.

If the system over which Romulus is applied already uses nested virtualization, the overhead of the technique is almost non-existent, otherwise the introduction of nested virtualization, which is a requirement of Romulus, will bring a considerable amount of overhead. Nevertheless, the overhead is still in line or better than that of other high-availability techniques and future developments to nested virtualization should lower the overhead.

Our technique is the only, as far as we know, that has been proven capable of tolerating both failures due to transient hardware faults and software faults. It is also one of the few techniques that have an implementation published under an open-source license. In our opinion, the biggest threat to the adoption of this technique in production systems is not the less-than-perfect VM recovery percentage, but rather the runtime performance overhead, and respective cost, that is associated to nested virtualization. Nevertheless, if we compare the overhead of our technique against other alternatives, our techniques possesses one strong point: no redundant hardware is required. For example, Remus [38], which provides tolerance against permanent hardware faults, requires a secondary host and incurs a performance overhead ranging between 30%-100% depending on the configuration and workload.

Another point to consider is that nested virtualization is gaining adoption in cloud computing, driven by user demand, as well as being a supporting technology for other techniques found in the literature, such as HyperFresh [9], which replaces a possibly corrupt hypervisor with a fresh instance before a failure takes place as a means to perform software rejuvenation, and TinyChecker [140], which protects VMs from a misbehaving hypervisor by monitoring the communication between them. As such, adding Romulus to a system that already uses nested virtualization carries almost no overhead.

7 AVAILABILITY-AS-A-SERVICE

This chapter presents the *Availability-as-a-Service* (AaaS) framework, which has the objective of increasing the availability of cloud computing by providing a robust framework that can be extended with various fault tolerance techniques that can be used by cloud providers and clients. A configurable set of modules that implement fault tolerance techniques and related functionalities allow AaaS to ensure the availability of the infrastructure.

We foresee that AaaS may be used by both cloud providers and cloud clients, albeit with different objectives. Cloud providers will be in charge of setting up the framework in their infrastructure, choosing which modules will be available and activated by default (system-wide), which reflects the cloud provider's conceived trade-off between availability and performance. Their goal in adopting this framework will be to offer a high-availability service, which can be marketed as a special product to their clients (*e.g.*, similarly to what many public cloud providers do with regards to shielded and confidential VMs [8, 33, 100, 130]), while reducing the costs (both upfront and recurring) required to do so. This goal is attainable using AaaS because existing alternatives for providing fault tolerance in virtualized or cloud systems usually require spare hardware (hence increasing acquisition costs), tend to be geographically distant and demand constant network and CPU usage to replicate state (thus increasing bandwidth and energy consumption), while still having a significant runtime overhead. AaaS manages to provide availability at the infrastructure-level and without being encumbered by these limitations.

Another driver for adoption from cloud providers can be the possibility to reduce SLA violations or to increase the offered uptime at a reduced cost to the provider. In this latter hypothetic scenario, the framework would be configured to prioritize performance during the majority of the time, with the possibility of selectively enabling some fault tolerance modules.

Cloud clients using AaaS will be able to explicitly define the active modules for each individual VM, even though they may simply use the cloud provider's default configuration. For the cloud client, this configurability enables prioritization of certain VMs (*e.g.*, VMs hosting mission-critical services), thus attaining a more fine-grained control over availability and performance and tailoring this trade-off based on the workload that the client desires to execute.

The framework is designed according to the following observations, which originate from empirical data obtained through the various

fault injection campaign experiments that have been performed and described in Chapters 4, 5 and 6.

- A node may fail in an abrupt manner, causing it to crash or hang and bring down all of the VMs running on it (*i.e.*, common-mode failures);
- An application running in a VM may crash or hang, while leaving the VM and the remaining virtualized system in a correct state;
- An application running in a VM may produce silent data corruption that can be stored locally or propagated outside of the system (*e.g.*, incorrect responses sent to the clients).

These failure modes can be caused by hardware and software faults and may be tolerated using fault tolerance as follows:

- Transient hardware faults and most software faults can be recovered by rebooting or resetting the affected component, as long as corruption has not propagated to other components;
- Permanent hardware faults must be handled using distributed fault tolerance techniques, such as VM migration across geographically distant availability zones, which are already common in cloud computing;
- Software faults that cannot be recovered only with state resetting, may be tolerated using code diversity.

According to our results, the majority of failures consist in crashes and hangs of one application or of all the VMs in the system, depending on the affected component and type of fault, however a small percentage of failures will lead to data corruption, which must be detected and handled before it propagates to the exterior of the system. Failures may affect applications running inside a VM, the entire VM or the hypervisor and its assisting VM – the PVM.

AaaS provides tolerance at the infrastructure-level, more specifically, it provides tolerance to a single node and all of the VMs and applications running on top of it. Nevertheless, existing fault tolerance techniques that use distributed approaches should be able to integrate seamlessly with AaaS. To demonstrate the integration between AaaS and other fault tolerance techniques, we analyze the usage of a straightforward watchdog and restart mechanism hosted in a remote machine that detects whenever a machine is unresponsive and restarts it using hardware support. The addition of a remote mechanism removes some of the advantages of having a local solution, but offers protection against failures due to rare faults that affect or propagate to the single point-of-failure, which in this case consists in the implementation that supports AaaS.

The sections of this chapter describe the AaaS framework, including a summary of the considered failure modes and how they may be tolerated, guidelines for hardening the software implementation supporting the framework and a default definition of the modules that may be used. Furthermore, a case study that analyzes how Romulus can be ported into the AaaS framework as a module is presented, as well as a case study showing how an external watchdog and reset mechanism can be used to cover additional failure modes of a system that follows the AaaS framework.

7.1 ARCHITECTURE

The AaaS framework must provide a way for cloud providers and clients to configure its behaviour and for modules to implement the functionalities that convey availability to the system. All of these operations must be done while ensuring that a fault in one of the components or any other unexpected behaviour has a limited impact in the system and that the addition of AaaS in itself does not overly reduce the availability or performance of the system. Hence, a careful design based on nested virtualization and separation between three levels of privilege was adopted, which is characterized by the guidelines that follow.

The first guideline mandates *the existence of a minimal software layer that is responsible for the essential virtualizating functionalities and for supporting AaaS* and which must be placed below any other layer (*i.e.*, it is the highest privileged layer). The software component that embodies this layer must provide memory virtualization, scheduling, nested virtualization and VM introspection support. We denominate it as *microvisor* due to its small size and hypervisor-like properties. If we compare a microvisor to a traditional hypervisor, the microvisor does not need to provide the device drivers, does not require a PVM, has a very simple toolstack and is transparent to the VMs that run above it, hence does not need a complex mechanism for communication with its VMs. These and other differences contribute to its small spatial and temporal footprint. The benefits of offloading the core functional requirements to the microvisor are a reduction in the amount of lines of code that execute in the most privileged level, thus reducing the amount and impact of software faults, as well as allowing the usage of formal verification methods to ensure its correctness.

Secondly, *modules of the framework should be isolated from other modules and from the microvisor* by executing at a privilege layer above that of the microvisor and using hardware mechanisms for this purpose, such as isolated memory areas. Modules provide the majority of the functionality to the AaaS framework and can be added or removed by the system administrator. Since they are the pluggable pieces of the framework, it is likely that modules will have higher software

defect density (*i.e.*, number of software faults per lines of code) than the microvisor, therefore they should be isolated as well as possible from the rest of the system.

The third guideline is that *modules and microvisor must provide a service interface that can be used by the cloud client, cloud provider, the microvisor or other modules to request services from other modules and from the framework*. Interaction among modules and the microvisor is essential for integration and maximizing synergies and a well-defined interface facilitates this task. The framework must also allow cloud providers and clients to configure the list of modules that is active in the system or for a specific VM.

From the above information, the design of AaaS can be derived, which will result in the architecture shown in Figure 32. In this figure Xen was used to exemplify the architecture, however AaaS supports any hypervisor.

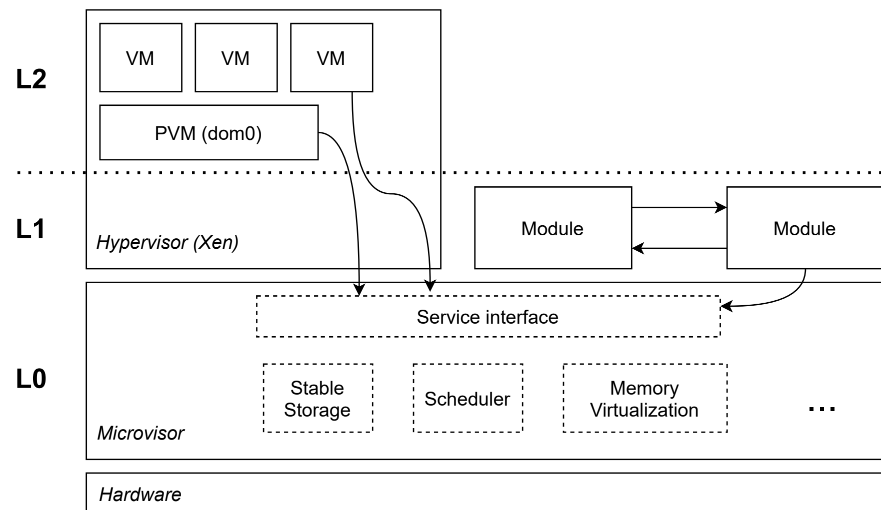


Figure 32: Architecture of the Availability-as-a-Service framework.

In the same manner that Romulus relied on nested virtualization (*i.e.*, multiple levels of virtualization, where one hypervisor virtualizes another hypervisor up to the desired nesting level) to support its operation, AaaS uses nested virtualization to isolate the microvisor from the remainder of the system, which includes the modules of AaaS and the VMs of the clients that are hosted in the node. In fact, any system that uses the AaaS framework is divided in three layers, representing the three levels of separation and privilege defined in the guidelines. The most privileged layer (L0) contains the microvisor, above which the modules and hypervisor are located. The microvisor must ensure isolation among hypervisor and modules to the extent allowed by the available hardware mechanisms, despite them sharing the same layer (L1). The least privileged layer (L2) contains the guest VMs that belong to the cloud clients.

Despite the existence of three isolated levels of virtualization, there is a need for communication between them. For example, a cloud client (in a VM) must be able to request the microvisor to apply specific modules to a VM, or, a module must be able to request and attain access to the memory of a VM. As already explained, these operations require calls to the service interfaces of the microvisor or of the modules.

To allow a module to have access to a VM's memory, which is essential for enabling some modules to implement their respective fault tolerance techniques, the AaaS framework must support the usage of VMI (Virtual Machine Introspection) [103] upon the request from the module. VMI enables read and write access to a VM's memory area without requiring alterations to the VM.

Certain modules will need to store permanent state that must persist across reboots of the modules. To support this requirement, the microvisor must provide a service to read and write from stable storage, which can further contains its own integrated fault tolerance mechanisms, such as data duplication for redundancy.

7.2 failure mode assumption

AaaS was designed as to support covering the highest percentage of failures that affect cloud computing while having a reduced performance overhead and cost. It can accomplish this objective, in part, because it is capable of tolerating most failure modes using fault tolerance at the infrastructure level, thus avoiding the costs and overheads associated with the cloud-level (*i.e.*, distributed) fault tolerance mechanisms that are currently employed by cloud providers.

Table 20 is built from the information collected in the previous chapters and lists the identified failure modes of a cloud computing node, as well as indicating at least one detection mechanism and recovery action that may be used to tolerate each failure mode. Failure modes are grouped according to the root cause (hardware or software faults) and the component that they affect: service (*i.e.*, the application executing inside the VM), VM or hypervisor (which includes the PVM that supports the hypervisor). Failures that occur in a component usually (but not always) do not affect components that reside at a lower level (*e.g.*, a failure in an application of a VM rarely affects the hypervisor), but the opposite is true (*i.e.*, usually failures of the hypervisor affect VMs and their applications). In other words, failures tend to propagate from the bottom up.

Most failure modes, including those that have been shown to account for the majority of failures, such as crashes of a VM or of the entire virtualized system (common-mode failures that affect all VMs co-located in the same node), are tolerable without the need for costly mechanisms, as long as the client is capable of detecting a timeout and

	FAILURE MODE	ROOT CAUSE	DETECTION	RECOVERY
Service	Crash	HW & SW faults	Timeout	Retry
	Hang		Timeout	Retry
	Timeout		Timeout	Retry
	Corruption		Redundancy	Checkpoint & Roll-back
VM	Crash	HW & SW faults	Timeout	Restart VM
Hypervisor	Crash Hang	Transient HW & SW faults	Timeout	Restart node
	Crash	Permanent HW faults	Timeout	Passive replication

Table 20: Failure modes of a cloud computing node.

retrying connection. These failures can be recovered using a mechanism that detects whether the service or infrastructure is unresponsive and performs a reboot of the VM or of the entire virtualized system. The drawback with this recovery approach is the considerable downtime associated with the restart process, however other more costly mechanisms can be used to tolerate these failure modes without downtime, such as performing VM replication across different nodes.

A similar failure mode, requests of the client that are not answered in the alloted time (*i.e.*, timeout), requires application-specific recovery mechanisms depending on the idempotency of the operation. If the operation is idempotent, retrying the operation is enough to recover from this failure (assuming that the service has recovered in the meantime), however if the operation is non-idempotent, a mechanism that ensures that an operation is not re-executed must be used, such as maintaining a history of responses.

The least likely failure mode, silent data corruption, is also the hardest to detect and can cause significant damage. These failures can be detected by the client whenever the expected response follows a well-defined format and the corrupted data violates this agreement, however corruption that only affects values (*e.g.*, the amount of money in a bank account is returned as 130€ instead of 1300€) can only be detected using redundancy, such as performing the same operation twice and comparing the result.

AaaS supports the integration of fault tolerance mechanisms that tolerate all failure modes shown in Table 20, except when the failures are caused by permanent hardware faults, which require distributed fault tolerance. These mechanisms may include detecting and restarting VMs that have entered hangs, duplicating VM execution to detect data corruption in I/O, maintaining a list of responses that have been sent to clients as to allow retry of non-idempotent operations, among others. Nevertheless, AaaS can be combined with classical distributed

fault tolerance techniques to cover a wider spectrum of failure modes, as will be shown in Section 7.7.

7.3 HARDENING THE AAAS FRAMEWORK

When introducing mechanisms to improve the dependability or availability of a system, a common approach relies on the addition of a new component that provides fault tolerance and which replaces the component that was the single point-of-failure (SPOF) of the system by taking its place instead. A possible example of this situation would be the addition of a voter mechanism in a TMR system, which becomes the new SPOF of the system. While at first glance it may seem like introducing such a component means adding more exposure to faults in a critical part of the system, that is not the case if the new component is robust and has a smaller codebase and runtime profile, thereby representing a net positive effect in the overall dependability of the system.

This observation is applicable also to the AaaS framework, since the framework requires its own software component for managing the modules, interacting with cloud provider and clients, monitoring system health and the various other functionalities that it implements. This software component must be robust, have a limited size that accounts only for the essential functionalities and be capable of tolerating unexpected faults and limiting the extent to which a failed sub-component (*e.g.*, a module) can affect the remaining part of the system. Furthermore, the microvisor is responsible for ensuring the temporal and spatial isolation among modules, VMs and itself. For these reasons, we propose that the AaaS framework should adhere to several principles found in literature from multiple research topics, ranging from microkernels to crash-only software and service architectures, which are detailed below.

1. *Every module must have its private memory area* that cannot be accessed under normal circumstances by the microvisor or other modules, in order to avoid unexpected operations that target these memory areas, which may occur due to latent software faults in the microvisor or modules. This choice ensures that a module cannot disturb the operation of the other modules or of the microvisor in any situation, as well as ensuring that a faulty microvisor cannot inadvertently corrupt the protected memory of the modules. Due to the need for communication among modules and microvisor, an exception to this rule must be allowed in the form of memory sharing grants;

2. *Modules communicate among themselves through a well-defined service interface*. This communication should be synchronous, blocking and every message should have a fixed-length (*e.g.*, a page size)

to avoid software faults due to improper handling of variable-sized messages. Synchronous communication enables simple implementation of mechanisms for detecting a hanged module and respective recovery measures;

3. *Modules and applications in VMs communicate with the microvisor through a well-defined service interface*, which exposes different functionalities depending on who is calling it. VMs have access to an *external interface* which allows configuration by the cloud clients of the modules that will be active. Modules have access to an *internal interface* which provides access to low-level functions that can include spawning, pausing and destroying VMs, requesting shared memory grants, among others. The implementation mechanism and properties (*e.g.*, blocking and synchronous) of the communication between microvisor and modules/VMs should be same that is used for communication between modules;

4. *Modules are advised to follow a crash-only approach*, when possible, which means that they should support being abruptly terminated at any moment in time and restarted while being able to resume correct operation. This property allows misbehaving modules to be terminated and rebooted, as well as allowing preventive rebooting of modules at regular intervals in order to perform software rejuvenation. Developing a module to be crash-only is not a trivial task, thus the AaaS framework still accepts modules that do not follow the crash-only principles;

5. *Persistent data required for a module's operation must be registered with the microvisor*. Modules may need to keep track of information which must remain available through reboots, in that case, modules are required to interact with the microvisor to request storage and retrieval of this information in a state store. The microvisor will then maintain a state store with this information and will use fault tolerance mechanisms, such as duplicated (or more) copies and comparison on read operations, to ensure the correctness of the information. Data that is temporary does not need to be stored by the microvisor and can remain in the module's address space, but will be deleted upon a module termination or reboot. Nevertheless, modules should strive to avoid usage of persistent data due to performance costs and to maximize the crash-only behaviour of the module;

6. *There must be a watchdog component, which monitors the state of every crash-only module and reboots any module that is deemed unresponsive during a pre-defined period of time*. This component performs periodic checks (*e.g.*, heartbeats) to every other module, through a well-defined service interface call, and requests a module re-

boot to the microvisor when it detects that a module has not responded during a considerable amount of time. This feature ensures that hanged or severely misbehaving modules are rebooted with minimal impact to system availability and depends on the crash-only capability of the modules. Simple heartbeats may not be sufficient to accurately detect the correctness of a module, thus more advanced and module-specific correctness checks may be used;

7. *Communication between modules must support timeouts and retries*. Given the fact that modules may be restarted or disabled at any point in time, or even fail, and since communication between modules is synchronous, there must be a timeout and a retry mechanism underpinning this type of communication. When calls to services represent idempotent operations (*i.e.*, operations that can be repeated multiple times with the same result), a mechanism that retries the call after a certain time without response should be used. This allows transparent handling of microreboots and temporary hang failures of a module. If the operation is non-idempotent, a retry cannot be performed, so a timeout should be used to return control to the caller module, which should be capable of taking corrective measures based on the situation;

8. *Each module possesses a manifest*, which specifies, among other things, the services that the module may request from the microvisor. The microvisor enforces this manifest and ensures that a software fault in a module has a limited destructive potential. Furthermore, the manifest should list all the services that a module exposes, their parameters, return value and idempotency;

9. *Modules must be able to request and obtain read-only or read-write access to a VM's memory space*. Access to a VM's memory space, which is provided through VMI, must depend on the acceptance by the microvisor of a module's request to have access to a limited area of memory. Requests that specify the type of access and the area of memory reduce the chance for VM memory corruption due to an unexpected behaviour from the module and allow the microvisor to ensure that only modules that need this type of access are able to obtain it.

7.4 MODULES

Modules provide the techniques that convey availability to the system. Without modules the AaaS framework would be hollow and fail to deliver on its objectives, despite the careful design that has been built to support the dependable operation and effective integration of the modules into the framework. In fact, the framework leverages the

advantages of each individual module and enhances their effect on the availability of the system by providing a base that is robust against faults and that enables synergies between modules and provides useful functionalities (*e.g*, persistent storage) that the modules can take advantage of.

Modules may directly provide availability (*e.g.*, by implementing a fault tolerance technique) or they may provide functionalities and resources that will be used by other modules to then provide availability. The modules that provide availability can do so in two ways: i) by reducing the MTTR, such as by providing fault tolerance mechanisms with short recovery times, or ii) by hardening the system and reducing the MTBF.

The list of modules that can be used in an AaaS framework is virtually infinite, as new modules can be developed and integrated into it. Nevertheless, we propose a list of base modules that tolerate most failure modes found to occur in cloud computing infrastructure. All the modules in this list contribute to the system's availability by reducing the MTTR of the system, instead of improving the MTBF.

M1 *Hypervisor duplication with local VM migration* – A module that spawns a second L1 hypervisor to support the local migration of (L2) VMs from a failed hypervisor to a new hypervisor, thus recovering a significant portion of VMs from failing due to a hardware or software fault in their hypervisor. This module refers to the fault tolerance technique described in Chapter 6, Romulus, which has not yet been ported to the AaaS framework. However, such an endeavor is feasible and will be described in the current chapter;

M2 *Local VM checkpointing & rollback* – A module that takes checkpoints of a VM state at a certain frequency, so that the state of the VM can be restored to a past and possibly correct state whenever a failure is detected, as to tolerate failures due to transient hardware faults that cause corruption in a VM;

M3 *VM duplication with output comparison and retry* – A module that reproduces a VM across two different hypervisor instances co-located on the same physical node and which compares the produced output (*e.g.*, on disk writes and network output) as to detect failures that cause discrepancies in the output. When a failure is detected the VM state may be returned to a previous state by integrating this module with the *Local VM checkpointing & rollback* module.

Table 21 indicates a possible association between each of the aforementioned modules and the identified failure modes of cloud computing that they can tolerate.

	FAILURE MODE	ROOT CAUSE	MODULE
Service	Crash	HW & SW faults	M2
	Hang		M2
	Timeout		M2
	Corruption		M2 + M3
VM	Crash	HW & SW faults	M2
Hypervisor	Crash	Transient HW & SW faults	M1, M2
	Hang		M1, M2
	Crash	Permanent HW faults	—

Table 21: Association between failure modes and modules.

The module M1 is capable of tolerating failures affecting the hypervisor that do not cause data corruption. To tolerate the failures where data corruption occurs, a combination of the modules M2 and M3 is required, in which module M3 detects the occurrence of corruption and then triggers module M2 to rollback the VM state to a previous copy. The module M2 is also capable of tolerating failures of services and VMs where there is no corruption. Failures of the infrastructure, including failures that affect the software component that supports the AaaS framework, cannot be tolerated locally and require distributed failure tolerance mechanisms, such as a remote watchdog and reset mechanism.

7.5 SERVICE INTERFACE

The integration of modules in the framework is a key aspect in its expandability. To accomplish effective integration, every module needs to specify a well-defined service interface that contains all functions that may be accessed by other modules, hence enabling composition and cooperation among modules. The microvisor also exposes its own service interface, which is split into an internal service interface, which can only be accessed by modules, and an external service interface, which can only be accessed from a VM (*i.e.*, it is available to cloud providers and clients).

7.5.1 *Microvisor Service Interface*

The microvisor exposes a well-defined internal service interface that can be accessed by any module to request management operations. This interface should extend after the fundamental interface that is shown in Table 22.

NAME	DESCRIPTION
Spawn VM	Spawns a new VM according to the passed configuration.
Destroy VM	Destroys a VM.
Pause VM	Pauses the execution of a running VM.
Resume VM	Resumes the execution of a paused VM.
Clone VM	Clones an existing VM instance into a new instance.
Request VM memory access	Requests a grant to access the memory of a VM.
Release VM memory access	Releases a grant that provided access to the memory of a VM.
Read VM memory	Uses VMI to read a memory portion of a VM, provided a grant has been given. Parameters should indicate the VM, memory address and size that should be read.
Write VM memory	Uses VMI to write a memory portion of a VM, provided a grant has been given. Parameters should indicate the VM, memory address, size and value to be written.
Hook Event	Interrupts VM execution and performs a callback to a module function whenever a specified event (*e.g.*, VMEXIT) takes place.
Trap memory access	Interrupts VM execution and performs a callback to a module function whenever a specified memory address is accessed.
Read CPU state	Returns the CPU state, including register values, of a CPU in a specified VM.
Reboot module	Destroys and restarts an instance of a module. For use by the watchdog component to reboot unresponsive modules.
Write to persistent store	Writes a key-pair value to a persistent store kept by the microvisor.
Read from persistent store	Reads a value given its key from a persistent store kept by the microvisor.

Table 22: Essential services of the microvisor internal service interface.

Most services deal with managing the L1 VMs running over the microvisor, accessing their memory (*e.g.*, using virtual machine introspection) or setting up callbacks on certain events. The functionalities included in this service interface were generalized from the knowledge about existing fault injection mechanisms that was obtained from literature review and the observations extracted from the experimental campaigns that have been presented in this thesis. The presented services are sufficient to support the operations that the modules presented in this chapter need to implement their fault tolerance mechanisms.

At the same time, the microvisor exposes an external interface to be used by cloud clients and providers to control and configure the modules that will be operating on a system-wide and per VM basis. Table 23 presents an example of the idealized external interface.

NAME	DESCRIPTION
Turn on module	Enables the operation of a specific module, system-wide or VM-specific.
Turn off module	Disables the operation of a specific module, system-wide or VM-specific.

Table 23: Example of the microvisor external service interface.

7.5.2 *Module Service Interface*

Modules have their own service interface which can be called by other modules. Given that modules may implement varied functionalities, it becomes difficult to define a base service interface. Nevertheless the interface shown in Table 24, which includes a heartbeat function for integration with the watchdog component of AaaS, should represent a subset of the functions included in any module's interface. However, non crash-only modules do not need to implement the heartbeat service call.

NAME	DESCRIPTION
Heartbeat	Entry point to be called by the watchdog component for verifying that the module is responsive.

Table 24: Example of the module service interface.

7.6 LIMITATIONS

The AaaS framework focuses on providing availability to cloud computing using fault tolerance at the infrastructure level. This approach

has several advantages in terms of lower resource usage (*e.g.*, no need for different physical machines), lower overhead (*e.g.*, no network usage) and simplicity. Nevertheless, it misses opportunities that could exist if a distributed approach had been considered. For example, since cloud providers manage multiple physical machines, some of which are idle or partly unused during large amounts of time, these machines could be used in a classic active-active or active-passive replication scheme, thus obtaining very low MTTR values and a high availability. During the development of AaaS we opted to focus on fault tolerance at the infrastructure level because it has been less explored than fault tolerance in distributed systems, has some already mentioned advantages and consolidates the work done in the thesis.

The addition of the microvisor and the dependency on nested virtualization are two other limitations of the AaaS framework. In one hand, nested virtualization has been shown to carry a non-negligible performance overhead, although we expect that this overhead may be lowered as nested virtualization becomes more mature. On the other hand, adding the microvisor means that a new component is being introduced in a critical part of the system (since the microvisor is located directly above the hardware, it represents a SPOF), thus increasing the system's exposure to failures caused by transient hardware faults and software faults. It is precisely due to this reason that it is essential that the microvisor has the smallest possible footprint and is thoroughly tested.

The usage of a service interface that cloud providers and clients can use to configure properties of the framework can also open the possibility for operator faults. However, these faults would mostly cause unintended configurations that affect the performance and availability levels of the system or of a VM.

7.7 CASE STUDY: INFRASTRUCTURE REBOOT USING AN EXTERNAL WATCHDOG

A node that follows the AaaS framework may still fail in a manner that the framework cannot tolerate. This may be the case if a fault in the software component that provides the implementation of AaaS (*i.e.*, the microvisor) is activated and causes its failure. In this case, recovery must be obtained using an external system.

In this section, an external secure watchdog and reset mechanism is described and its performance evaluated. The mechanism is composed by a watchdog component that monitors the state of a remote node using integrity tests, and by a reset mechanism, which is triggered externally by a watchdog timer when a failure is detected. The employed integrity tests can vary according to the user needs, but for our experiments we used tests made via SSH that exercised simple tasks of the target system. A first attempt at designing integrity tests that

used ping requests was done, however it was observed that often the pings would incorrectly indicate that the target system was healthy despite it being in a hanged state. This observation emphasizes the need to execute integrity tests on the target system, rather than using simple but ineffective `ping` mechanisms or heartbeats.

Once the watchdog determines that the hypervisor is in a hanged state, it issues a remote command to physically reset the hardware. This is achieved by using Intel's AMT technology, which supports remote power cycling. Using Intel's Ethernet interface, network packets are inspected by the hardware immediately at reception. Specific network packets, recognized by Intel AMT hardware, lead the hardware to physically reset the machine.

In order to prevent unintended remote restarts, which could potentially lead to security vulnerabilities, Intel AMT supports Transport Layer Security (TLS). Power cycling commands, used to reset the physical machine, may therefore be encrypted with a key shared between the watchdog and the physical machine running the virtualization server.

In order to validate the proposed external watchdog, we conducted a fault injection campaign to measure the efficiency – the proportion of failures that are correctly recovered – and the latency – the time it takes to reset. As the target system of the reset, a virtualized node running one VM was used. However, other types of systems, including systems that follow the AaaS framework, can be reset using this mechanism. Logically, the latency will depend on the physical and logical properties of each system and may vary slightly from the results shown in this section.

A HTTP workload where multiple clients performed requests to an Apache server during 2 minutes was used. Fault injection in CPU registers (more precisely, *rip*, *rbx* and *rsp*) while processes of the PVM were executing was employed to accelerate the occurrence of failures of the infrastructure.

After injection, we left the system running and the external watchdog recovered the hypervisor correctly in 100% of the cases in which the system hanged. The time to recover was monitored and, as soon as possible, an SSH connection was established to the hypervisor, to run the integrity tests, and one minute of HTTP requests were issued to the Web server running inside the virtual machine. The HTTP service was also resumed correctly in all cases.

Table 25 shows the evaluation results, which demonstrate the effectiveness and recovery latency of the mechanism.

The external watchdog recovered the system in all 203 experiments in which the system hanged. The recovery time for the hypervisor is on average 31.9 seconds, with a worst case of 34 seconds; the subsequent recovery time for a virtual machine is 75.2 seconds on average, with a worst case of 103 seconds.

SYSTEM STATE		HYPERVISOR RECOVERY			VM RECOVERY		
		MIN	MAX	AVG	MIN	MAX	AVG
Hang	203	30 s	34 s	31.9 s	54 s	103 s	75.2 s
No effect	3						

Table 25: Evaluation of the external watchdog and reset.

7.8 CASE STUDY: INTEGRATING ROMULUS INTO AAAS

Although the proof-of-concept implementation of Romulus was developed as a standalone project, Romulus (which was presented in Chapter 6) is an ideal candidate to be integrated into the AaaS framework as one of the modules that can be enabled to protect the infrastructure. In this section, we specify the essential properties that are required to encapsulate Romulus into a module of AaaS, such as the service interface and the manifest.

7.8.1 *Service Interface*

Other than the heartbeat function that should be part of the service interface of any module (see Table 24), a module implementing Romulus should possess also the services described in Table 26.

NAME	DESCRIPTION
Monitor guest VM	Enables monitoring of a specific guest (L2) VM.
Stop monitoring guest VM	Disables monitoring of a specific guest (L2) VM.
Trigger recovery action	Starts the process of migrating all monitored L2 VMs from one hypervisor to another.
Configure detection interval	Enables the configuration of the amount of seconds that the embedded timeout mechanism will wait before triggering recovery action.

Table 26: Service interface of a module that implements Romulus.

Romulus has been presented and used as a technique that attempts to recovers all VMs running in a virtualized system upon failure of the hypervisor. Nevertheless it can also be used to recover a specific and limited set of L2 VMs, which enables selective protection of the most critical VMs and a trade-off between performance, coverage and MTTR. To do so, the module exposes *Monitor guest VM* and *Stop monitoring guest VM* services that can be called by the cloud client.

Error detection mechanisms have an important role in ensuring effective fault tolerance when using Romulus. A detection and triggering mechanism may come integrated as part of the module, for example, in the form of a simple ping mechanism similar to what was used in Chapter 6, or an external module may be used to provide the error detection mechanism, thus taking advantage of the flexibility that the AaaS framework supports. An embedded error detection mechanism can be configured using the *Configure detection interval* call, whereas the *Trigger recovery action* service can be used by external mechanisms to trigger recovery action.

7.8.2 *Manifest*

Modules need to request functionality provided by the microvisor, however not all the functionality available should be used. To avoid unintended consequences due to software bugs in the modules, which may perform incorrect calls to the microvisor, a manifest must be defined in the module, that states which functions it may request from the hypervisor. In the case of a module implementing Romulus, it requires the following services of the microvisor (see Table 22 for the complete list of the default service interface of the microvisor).

- *Spawn VM* – The module needs to be able to spawn a new L1 hypervisor to accommodate the VMs of the failed hypervisor;
- *Pause VM* and *Resume VM* – The module can use pause/resume functionality to reduce performance overhead due to the idle L1 hypervisor, however the essential functionality required by Romulus that these functions provide is being able to pause the failed L1 hypervisor before the failure propagates to the L2 VMs;
- *Request VM memory access*, *Read VM memory*, *Write VM memory* – Romulus depends heavily on virtual machine introspection to extract the state of the L2 VMs;
- *Hook Event* – The proof-of-concept implementation of Romulus uses the VMEXIT handler of the microvisor to monitor and obtain part of the latest CPU state of the L2 VMs. The same functionality can be implemented in AaaS by associating a callback through this service function;
- *Read CPU state* – The remaining part of the CPU state that is not obtained by hooking into the VMEXIT handler is already available in the microvisor, therefore it can be easily obtained by our module by calling this service function.

As long as these services are available and provided by the microvisor, it is possible to convert Romulus into a module of AaaS.

7.9 RELATED WORK

Several works have dealt with design and implementation of robust and dependable software and have produced fault tolerance techniques and software design patterns to this effect. The specification of the Availability-as-a-Service framework takes some of the ideas found behind crash-only software, microrebooting, microkernels and operating systems.

In fact, many of the presented rules have been proposed or used in other works. For example, the Minix operating system [66] uses fixed-length messages and synchronous message passing for communication between processes. Moreover, the concept of service interfaces can be equated to system calls of operating systems, although with the addition of a manifest.

Watchdog components that monitor the operation of other components and trigger recovery actions after a timeout has been elapsed are also far from novel. For example, such a component, which was coined as 'reincarnation server', was used in an effort to provide fault tolerance against failures of device drivers [64]. Given that in many situations there is a need to store non-volatile information (*i.e.*, information that cannot be lost between reboots of the component), data stores have traditionally been used and backed by a stable storage that can tolerate the considered failure modes and provide atomic transactional access to the data [79].

Crash-only software [21] defends that software should be grouped into well-defined self-contained components that can only be stopped and started in one way – by crashing and restarting it. In order to convert a program to this paradigm, there must be a separation between program logic and data logic, therefore all important non-volatile state should be kept in a data store. Furthermore, since components may be rebooted at any time, communication between modules has to be ready to handle this fact, through the usage of timeouts and support for retrying idempotent operations. The concept of resource leases was also applied due to the need to manage resources that might become stale after a component is rebooted.

Microkernels [66, 113] are kernels of minimal size that implement only the basic functionalities of an operating system, such as scheduling and memory management, and defer the remaining tasks to other components that usually execute in less privileged layers. This type of kernel is specially well-adapted to be used in a crash-only setting due to the componentization and compartmentalization that it implies and which facilitates failure isolation. Furthermore, its small size reduces the likelihood of software faults and enables it to be formally verified, as was successfully performed with the seL4 microkernel [75].

Microrebooting [22] uses the ideas behind crash-only software to propose a technique for cheap and quick rebooting of a small compo-

nent inside of a system. Given the inexpensive cost of performing a microreboot, this operation can be performed as a first step of recovery, whenever the first signs of a failure appear, or even preventively, as a mechanism of software rejuvenation. Microrebooting has been loosely applied before to virtualized systems as a technique to recover VMs from a failing hypervisor [83].

7.10 SUMMARY

Availability is of the utmost importance to cloud computing, yet existing fault tolerance techniques, in particular those that are currently available in public clouds (*e.g.*, availability zones, physical redundancy), do not cover many failure modes, are costly to cloud providers and clients and are not transparent to the client. This chapter has presented Availability-as-a-Service (AaaS), a framework that harnesses the integration of different modules that provide fault tolerance with the objective of increasing the availability provided by the cloud, all the while reducing operational costs, being configurable by cloud providers and clients, and being generic. AaaS is applied at the infrastructure level but can be combined with distributed fault tolerance techniques, such as an external watchdog and reset mechanism, to cover the failure modes that cannot be tolerated using only a single node, such as failures caused by permanent hardware faults that require a damaged hardware component to be replaced.

To improve its robustness, AaaS itself follows principles proposed in literature, such as enforcing memory isolation among components, avoiding dynamic memory allocation, or using manifests to limit failure propagation between components. The design of AaaS calls for the addition of a lean layer between the hardware and the hypervisor of the node. This microvisor manages the basic operations of the system, such as CPU scheduling and memory management, provides an entry point for cloud providers and clients to configure AaaS and supports the execution of the hypervisor and its VMs through the usage of nested virtualization. Due to being a possible single point of failure and having control over the rest of the system, the microvisor should have a small temporal and spatial footprint, thus reducing its exposure to transient hardware and software faults, and should be thoroughly tested, eventually even using formal methods.

Fault tolerance is provided by modules that implement contained fault tolerance techniques and which can be added, removed, enabled and disabled at will by the cloud provider. Modules can cooperate by combining their functionalities, which is accomplished through a well-defined service interface that every module should provide and from where other modules can call the available functions.

In summary, AaaS offers cloud providers and availability-focused clients the possibility to take advantage of fault tolerance mechanisms

to reinforce availability levels of the entire system or of a specific set of VMs. This framework was designed using information obtained from the literature review and experimental campaigns conducted during this thesis, thus consolidating all of this thesis' contributions into a single consistent unit.

8 CONCLUSION

In order for the cloud to be trusted and accepted as a platform that can support mission-critical workloads, the level of dependability that can be provided to cloud clients must, at least, match that of dedicated infrastructures. Accordingly, one of the objectives of this thesis is to improve cloud computing dependability, which we accomplished by proposing fault tolerance mechanisms that were designed and validated using empirical methods.

Existing techniques for fault tolerance and increasing dependability in other fields resort to highly redundant setups, which contain multiple copies of the same hardware and may even use various software implementations. Although such techniques are greatly effective at tolerating the large majority of failures, they are also extremely costly, which directly negates one of the advantages promoted by cloud computing – lower costs due to resource sharing. On the other hand, fault tolerance mechanisms commonly used in cloud computing rely mostly on distinct geographical separations (*e.g.*, availability zones), VM migration across different physical hosts in the same datacenter, or simple techniques that rely on rebooting VMs and physical systems to eliminate possibly corrupted or stale state, along with memory and resource leaks. In general, these techniques are insufficient for tolerating most of the failures caused by software faults and transient hardware faults, thus there is a need for more advanced fault tolerance techniques that provide dependability and availability, cover a wider range of failures, while carrying a limited performance overhead.

However, if effective and efficient fault tolerance for cloud computing is to be attained, redundancy must be applied strategically, by targeting the components that are more likely to be affected by faults, more likely to cause failures that affect more clients, and the failure modes that are more common to occur or which pose a bigger risk to dependability. In short, designing fault tolerance mechanisms requires an evaluation of cloud computing dependability, which is precisely the second objective of this thesis.

Knowledge about the dependability of cloud computing and how it fails must be extracted from representative failure data. However, such an endeavor will take years to complete unless measures are taken to accelerate this process. In this thesis, an experimental methodology based around fault injection was used to accelerate the obtention of representative failure data that was needed to characterize cloud computing dependability. One contribution of the thesis is the development of the ucXception framework and its tools for performing

fault injection in cloud computing and virtualized systems. There was the necessity to create the fault injection tools that would be used for the experimental evaluation, since existing tools were not adequate to be used in cloud computing, as they did not support injection into the components required for virtualization, or did not support the desired fault models.

Using the ucXception framework, faults representing two of the most significant threats to cloud computing – transient hardware faults (*i.e.*, soft errors) and software faults – were injected in a virtualized system similar to those used in cloud computing. These types of faults are expected to increase in future years, due to technological advances and more complex and bigger cloud datacenters and cloud-related software, and have been shown to cause failure modes that are specially hard to handle (*e.g.*, silent data corruption). The observations obtained using fault injection show that the hypervisor and privileged virtual machine are specially at risk of causing failures that affect multiple clients at once and may also, albeit unlikely, cause failures that lead to data corruption. These observations constitute new insights into the importance that the components that provide virtualization have in the dependability of a virtualized system and, by consequence, in cloud computing, as well as cautioning against the usage of fault tolerance techniques that depend on redundancy across VMs that are hosted over the same hardware, since both VMs may be affected by common-mode failures. Furthermore, the results from fault injection confirm that virtualization is largely effective at isolating faulty VMs from the rest of the system.

To deal with failures of the hypervisor, we proposed a fault tolerance technique called Romulus, which migrates the VMs running in a failed hypervisor to a co-located hypervisor, thus enabling the VMs to continue executing in a healthy hypervisor with minimal downtime. Romulus was designed after we used fault injection to show that many times VMs are not affected after hypervisor failure, therefore their state remains correct and they can continue execution in a different hypervisor. Up to this point, the consensus was that whenever the hypervisor failed, no VM could be recovered. We stood for the opposite hypothesis, which argued that at least sometimes VMs would be left in a working, non-corrupted state after hypervisor failure. Romulus represents a novel fault tolerance technique designed specifically for virtualized and cloud systems and one of the few, if not the only, technique capable of tolerating failures caused by transient hardware faults and software faults without requiring redundant hardware. A proof-of-concept implementation was developed and evaluated using fault injection, thereby confirming that Romulus can recover at least part of the VMs affected by a hypervisor failure in most cases.

We also proposed the Availability-as-a-Service (AaaS) framework as a contribution that enables increasing the availability of cloud comput-

ing by providing tolerance against the most common failure modes in an extensible manner and in which cloud providers and clients can configure the desired fault tolerance mechanisms that are active. AaaS combines the need for availability by part of the cloud clients that want to execute their mission critical workloads and cloud providers that have SLAs to adhere to, with the optimization of resource usage that cloud providers strive for. The AaaS framework provides tolerance at the infrastructure-level using fault tolerance mechanisms that operate locally, however it can be combined with distributed mechanisms to tolerate failures that affect the infrastructure or the software component that supports the AaaS framework and which cannot be tolerated with local mechanisms.

Overall, this thesis contributes to a better characterization of the dependability of cloud computing, as well as producing a framework for carrying out fault injection campaigns in cloud computing and two well-defined proposals to improve the dependability of cloud computing at the infrastructure level. Possible future work resides in continuing to develop Availability-as-a-Service, as well as further improving Romulus and its proof-of-concept implementation. Furthermore, Romulus can be extended to take advantage of existing fault tolerance techniques, such as checkpointing and rollback, as to significantly improve its recovery effectiveness, at the expense of losing some performance.

[1] Keith Adams and Ole Agesen. “A Comparison of Software and Hardware Techniques for X86 Virtualization.” In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA: Association for Computing Machinery, 2006, 2–13. ISBN: 1595934510. DOI: 10.1145/1168857.1168860. URL: https://doi.org/10.1145/1168857.1168860.

[2] *Amazon Compute Service Level Agreement*. July 2020. URL: https://aws.amazon.com/compute/sla/.

[3] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage. “Steward: Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks.” In: *IEEE Transactions on Dependable and Secure Computing* 7.1 (2010), pp. 80–93. DOI: 10.1109/TDSC.2008.53.

[4] D. G. Andersen and S. Swanson. “Rethinking Flash in the Data Center.” In: *IEEE Micro* 30.4 (2010), pp. 52–54. DOI: 10.1109/MM.2010.71.

[5] J. Arlat, A. Costes, Y. Crouzet, J. C. Laprie, and D. Powell. “Fault injection and dependability evaluation of fault-tolerant systems.” In: *IEEE Transactions on Computers* 42.8 (1993), pp. 913–923. DOI: 10.1109/12.238482.

[6] J. Arlat, Y. Crouzet, and J. . Laprie. “Fault injection for dependability validation of fault-tolerant computing systems.” In: *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*. June 1989, pp. 348–355. DOI: 10.1109/FTCS.1989.105591.

[7] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. “Basic concepts and taxonomy of dependable and secure computing.” In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2.

[8] Azure Confidential Compute | Microsoft Azure. *Microsoft*. https://azure.microsoft.com/en-us/solutions/confidential-compute/. Accessed: 2020-11-25.

[9] Hardik Bagdi, Rohith Kugve, and Kartik Gopalan. “HyperFresh: Live Refresh of Hypervisors Using Nested Virtualization.” In: APSys ’17. Mumbai, India: Association for Computing Machinery, 2017. ISBN: 9781450351973. DOI: 10.1145/3124680.3124734. URL: https://doi.org/10.1145/3124680.3124734.

[10] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. "An Analysis of Data Corruption in the Storage Stack." In: *ACM Trans. Storage* 4.3 (Nov. 2008). ISSN: 1553-3077. DOI: 10.1145/1416944.1416947. URL: https://doi.org/10.1145/1416944.1416947.

[11] Raul Barbosa, Frederico Cerveira, Luís Gonçalo, and Henrique Madeira. "Emulating Representative Software Vulnerabilities Using Field Data." In: *Computing* 101.2 (Feb. 2019), 119–138. ISSN: 0010-485X. DOI: 10.1007/s00607-018-0657-y. URL: https://doi.org/10.1007/s00607-018-0657-y.

[12] Raul Barbosa, Johan Karlsson, Henrique Madeira, and Marco Vieira. "Fault Injection." In: *Resilience Assessment and Evaluation of Computing Systems*. Ed. by Katinka Wolter, Alberto Avritzer, Marco Vieira, and Aad van Moorsel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 263–281. ISBN: 978-3-642-29032-9. DOI: 10.1007/978-3-642-29032-9_13. URL: https://doi.org/10.1007/978-3-642-29032-9_13.

[13] Raul Barbosa, Jonny Vinter, Peter Folkesson, and Johan Karlsson. "Assembly-Level Pre-injection Analysis for Improving Fault Injection Efficiency." In: *Proceedings of the EDCC 5: 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 246–262. ISBN: 978-3-540-32019-7. DOI: 10.1007/11408901_19. URL: http://dx.doi.org/10.1007/11408901_19.

[14] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the Art of Virtualization." In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), 164–177. ISSN: 0163-5980. DOI: 10.1145/1165389.945462. URL: https://doi.org/10.1145/1165389.945462.

[15] Luiz Andr Barroso, Jimmy Clidaras, and Urs Hlzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 2nd. Morgan & Claypool Publishers, 2013. ISBN: 1627050094.

[16] Salman A. Baset. "Cloud SLAs: Present and Future." In: *SIGOPS Oper. Syst. Rev.* 46.2 (July 2012), 57–66. ISSN: 0163-5980. DOI: 10.1145/2331576.2331586. URL: https://doi.org/10.1145/2331576.2331586.

[17] R. C. Baumann. "Radiation-induced soft errors in advanced semiconductor technologies." In: *IEEE Transactions on Device and Materials Reliability* 5.3 (Sept. 2005), pp. 305–316. ISSN: 1530-4388. DOI: 10.1109/TDMR.2005.853449. URL: http://dx.doi.org/10.1109/TDMR.2005.853449.

[18] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. "The Turtles Project: Design and Implementation of Nested Virtualization." In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, 423–436.

[19] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. "Accelerating Two-Dimensional Page Walks for Virtualized Systems." In: *SIGOPS Oper. Syst. Rev.* 42.2 (Mar. 2008), 26–35. ISSN: 0163-5980. DOI: 10.1145/1353535.1346286. URL: https://doi.org/10.1145/1353535.1346286.

[20] R. Birke, I. Giurgiu, L. Y. Chen, D. Wiesmann, and T. Engbersen. "Failure Analysis of Virtual and Physical Machines: Patterns, Causes and Characteristics." In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2014, pp. 1–12. DOI: 10.1109/DSN.2014.18.

[21] George Candea and Armando Fox. "Crash-Only Software." In: *HotOS*. Vol. 3. 2003, pp. 67–72.

[22] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. "Microreboot — A Technique for Cheap Recovery." In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, p. 3.

[23] Ignacio Cano, Srinivas Aiyar, and Arvind Krishnamurthy. "Characterizing Private Clouds: A Large-Scale Empirical Analysis of Enterprise Clusters." In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. SoCC '16. Santa Clara, CA, USA: Association for Computing Machinery, 2016, 29–41. ISBN: 9781450345255. DOI: 10.1145/2987550.2987584. URL: https://doi.org/10.1145/2987550.2987584.

[24] J. Carreira, H. Madeira, and J. G. Silva. "Xception: a technique for the experimental evaluation of dependability in modern computers." In: *IEEE Transactions on Software Engineering* 24.2 (1998), pp. 125–136. DOI: 10.1109/32.666826.

[25] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. "Long-Term SLOs for Reclaimed Cloud Computing Resources." In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. Seattle, WA, USA: Association for Computing Machinery, 2014, 1–13. ISBN: 9781450332521. DOI: 10.1145/2670979.2670999. URL: https://doi.org/10.1145/2670979.2670999.

[26] V. Chandra and R. Aitken. "Impact of Technology and Voltage Scaling on the Soft Error Susceptibility in Nanoscale CMOS." In: *2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*. Oct. 2008, pp. 114–122.

[27] C. H. Chen, D. Blaauw, D. Sylvester, and Z. Zhang. "Design and Evaluation of Confidence-Driven Error-Resilient Systems." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.8 (Aug. 2014), pp. 1727–1737. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2013.2277351. URL: http://dx.doi.org/10.1109/TVLSI.2013.2277351.

[28] Liming Chen and A. Avizienis. "N-version programming: A fault-tolerance approach to reliability of software operation." In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing*. Los Alamitos, CA, USA: IEEE Computer Society, June 1995, p. 113. DOI: 10.1109/FTCSH.1995.532621. URL: https://doi.ieeecomputersociety.org/10.1109/FTCSH.1995.532621.

[29] David Chisnall. *The definitive guide to the xen hypervisor*. Pearson Education, 2008.

[30] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. "An Empirical Study of Operating Systems Errors." In: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. SOSP '01. Banff, Alberta, Canada: Association for Computing Machinery, 2001, 73–88. ISBN: 1581133898. DOI: 10.1145/502034.502042. URL: https://doi.org/10.1145/502034.502042.

[31] Marcello Cinque and Antonio Pecchia. "On the injection of hardware faults in virtualized multicore systems." In: *Journal of Parallel and Distributed Computing* 106 (2017), pp. 50 –61. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2017.03.004. URL: http://www.sciencedirect.com/science/article/pii/S0743731517300849.

[32] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. "Live Migration of Virtual Machines." In: *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings*. Ed. by Amin Vahdat and David Wetherall. USENIX, 2005. URL: http://www.usenix.org/events/nsdi05/tech/clark.html.

[33] Confidential Computing | Google Cloud. *Google*. https://cloud.google.com/confidential-computing. Accessed: 2020-11-25.

[34] Intel Corporation. *Intel PCM*. https://github.com/opcm/pcm. Accessed: 2021-02-01. 2019.

[35] Pedro Costa, João Gabriel Silva, and Henrique Madeira. "Practical and representative faultloads for large-scale software systems." In: *Journal of Systems and Software* 103 (2015), pp. 182–197. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2015.02.001. URL: http://www.sciencedirect.com/science/article/pii/S0164121215000357.

[36] D. Cotroneo, L. De Simone, P. Liguori, and R. Natella. "ProFIPy: Programmable Software Fault Injection as-a-Service." In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2020, pp. 364–372. DOI: 10.1109/DSN48063.2020.00052.

[37] Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Nematollah Bidokhti. "How Bad Can a Bug Get? An Empirical Analysis of Software Failures in the OpenStack Cloud Computing Platform." In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, 200–211. ISBN: 9781450355728. DOI: 10.1145/3338906.3338916. URL: https://doi.org/10.1145/3338906.3338916.

[38] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. "Remus: High availability via asynchronous virtual machine replication." In: *Proceedings of the 5th USENIX symposium on networked systems design and implementation*. San Francisco. 2008, pp. 161–174.

[39] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-Efficient and QoS-Aware Cluster Management." In: *SIGPLAN Not.* 49.4 (Feb. 2014), 127–144. ISSN: 0362-1340. DOI: 10.1145/2644865.2541941. URL: https://doi.org/10.1145/2644865.2541941.

[40] A. Dixit and A. Wood. "The impact of new technology on soft error rates." In: *2011 International Reliability Physics Symposium*. Apr. 2011, 5B.4.1–5B.4.7. DOI: 10.1109/IRPS.2011.5784522.

[41] YaoZu Dong, Wei Ye, YunHong Jiang, Ian Pratt, ShiQing Ma, Jian Li, and HaiBing Guan. "COLO: COarse-Grained LOck-Stepping Virtual Machines for Non-Stop Service." In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: Association for Computing Machinery, 2013. ISBN: 9781450324281. DOI: 10.1145/2523616.2523630. URL: https://doi.org/10.1145/2523616.2523630.

[42] J. A. Duraes and H. S. Madeira. "Emulation of Software Faults: A Field Data Study and a Practical Approach." In: *IEEE Transactions on Software Engineering* 32.11 (Nov. 2006), pp. 849–867. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.113.

[43] J. Duraes and H. Madeira. "Characterization of operating systems behavior in the presence of faulty drivers through software fault emulation." In: *2002 Pacific Rim International Symposium on Dependable Computing, 2002. Proceedings.* 2002, pp. 201–209. DOI: 10.1109/PRDC.2002.1185639.

[44] Michael Eischer and Tobias Distler. "Resilient Cloud-based Replication with Low Latency." In: *CoRR* abs/2009.10043 (2020). arXiv: 2009.10043. URL: https://arxiv.org/abs/2009.10043.

[45] Jon Elerath. "Hard-Disk Drives: The Good, the Bad, and the Ugly." In: *Commun. ACM* 52.6 (June 2009), 38–45. ISSN: 0001-0782. DOI: 10.1145/1516046.1516059. URL: https://doi.org/10.1145/1516046.1516059.

[46] D Faggioli. *Tracing with Xentrace and Xenalyze*. https://blog. xenproject. org/2012/09/27/tracing-with-xentrace-and-xenalyze. 2012.

[47] Q. Feng, J. Han, Y. Gao, and D. Meng. "Magicube: High Reliability and Low Redundancy Storage Architecture for Cloud Computing." In: *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*. 2012, pp. 89–93.

[48] R. Figueiredo, P. A. Dinda, and J. Fortes. "Guest Editors' Introduction: Resource Virtualization Renaissance." In: *Computer* 38.5 (2005), pp. 28–31. DOI: 10.1109/MC.2005.159.

[49] Robert W Floyd. "Assigning meanings to programs." In: *Mathematical aspects of computer science* 19.19-32 (1967), p. 1.

[50] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. "Safe hardware access with the Xen virtual machine monitor." In: *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*. 2004, pp. 1–1.

[51] Tal Garfinkel and Mendel Rosenblum. "A Virtual Machine Introspection Based Architecture for Intrusion Detection." In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society, 2003.

[52] P. Garraghan, I. S. Moreno, P. Townend, and J. Xu. "An Analysis of Failure-Related Energy Waste in a Large-Scale Cloud Environment." In: *IEEE Transactions on Emerging Topics in Computing* 2.2 (2014), pp. 166–180. DOI: 10.1109/TETC.2014.2304500.

[53] P. Garraghan, R. Yang, Z. Wen, A. Romanovsky, J. Xu, R. Buyya, and R. Ranjan. "Emergent Failures: Rethinking Cloud Reliability at Scale." In: *IEEE Cloud Computing* 5.5 (2018), pp. 12–21. DOI: 10.1109/MCC.2018.053711662.

[54] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. "EDFI: A Dependable Fault Injection Tool for Dependability Benchmarking Experiments." In: *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*. 2013, pp. 31–40. DOI: 10.1109/PRDC.2013.12.

[55] Sebastien Godard. *SYSSTAT*. http://sebastien.godard.pagesperso-orange.fr/. Accessed: 2021-02-01. 2019.

[56] R. P. Goldberg. "Survey of virtual machine research." In: *Computer* 7.6 (1974), pp. 34–45. DOI: 10.1109/MC.1974.6323581.

[57] J Gray and C van Ingen. *Empirical Measurements of Disk Failure Rates and Error Rates*. Tech. rep. cs.DB/0701166. MSR-TR-2005-166. Jan. 2007. URL: https://cds.cern.ch/record/1013594.

[58] Jim Gray. "Why Do Computers Stop and What Can Be Done About It?" In: *Fifth Symposium on Reliability in Distributed Software and Database Systems, SRDS 1986, Los Angeles, California, USA, January 13-15, 1986, Proceedings*. IEEE Computer Society, 1986, pp. 3–12.

[59] R. L. Grossman. "The Case for Cloud Computing." In: *IT Professional* 11.2 (2009), pp. 23–27. DOI: 10.1109/MITP.2009.40.

[60] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. "Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages." In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. SoCC '16. Santa Clara, CA, USA: Association for Computing Machinery, 2016, 1–16. ISBN: 9781450345255. DOI: 10.1145/2987550.2987583. URL: https://doi.org/10.1145/2987550.2987583.

[61] Haryadi S. Gunawi et al. "What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems." In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. Seattle, WA, USA: Association for Computing Machinery, 2014, 1–14. ISBN: 9781450332521. DOI: 10.1145/2670979.2670986. URL: https://doi.org/10.1145/2670979.2670986.

[62] S. Hagen, M. Seibold, and A. Kemper. "Efficient verification of IT change operations or: How we could have prevented Amazon's cloud outage." In: *2012 IEEE Network Operations and Management Symposium*. 2012, pp. 368–376. DOI: 10.1109/NOMS.2012.6211920.

[63] Steven Hand, Andrew Warfield, and Keir Fraser. "Hardware Virtualization with Xen." In: *login Usenix Mag.* 32.1 (2007). URL: https://www.usenix.org/publications/login/february-2007-volume-32-number-1/hardware-virtualization-xen.

[64] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. "Failure Resilience for Device Drivers." In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 2007, pp. 41–50. DOI: 10.1109/DSN.2007.46.

[65] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. "Fault isolation for device drivers." In: *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. 2009, pp. 33–42. DOI: 10.1109/DSN.2009.5270357.

[66] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. "MINIX 3: A Highly Reliable, Self-Repairing Operating System." In: *SIGOPS Oper. Syst. Rev.* 40.3 (July 2006), 80–89. ISSN: 0163-5980. DOI: 10.1145/1151374.1151391. URL: https://doi.org/10.1145/1151374.1151391.

[67] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. "Gray Failure: The Achilles' Heel of Cloud-Scale Systems." In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS '17. Whistler, BC, Canada: Association for Computing Machinery, 2017, 150–155. ISBN: 9781450350686. DOI: 10.1145/3102980.3103005. URL: https://doi.org/10.1145/3102980.3103005.

[68] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. "Software rejuvenation: analysis, module and applications." In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*. 1995, pp. 381–390. DOI: 10.1109/FTCS.1995.466961.

[69] ITRS. *International Technology Roadmap for Semiconductors*. 2013.

[70] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. "Are Disks the Dominant Contributor for Storage Failures? A Comprehensive Study of Storage Subsystem Failure Characteristics." In: *ACM Trans. Storage* 4.3 (Nov. 2008). ISSN: 1553-3077. DOI: 10.1145/1416944.1416946. URL: https://doi.org/10.1145/1416944.1416946.

[71] Rolf Johansson. *On Single Event Phenomena in Microprocessors*. 1993.

[72] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. "FERRARI: a flexible software-based fault and error injection system." In: *IEEE Transactions on Computers* 44.2 (Feb. 1995), pp. 248–260. ISSN: 0018-9340. DOI: 10.1109/12.364536.

[73] W. . Kao, R. K. Iyer, and D. Tang. "FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults." In: *IEEE Transactions on Software Engineering* 19.11 (Nov. 1993), pp. 1105–1118. ISSN: 0098-5589. DOI: 10.1109/32.256857.

[74] Peter Kelemen. "Silent corruptions." In: *8th Annual Workshop on Linux Clusters for Super Computing*. 2007.

[75] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. "Comprehensive Formal Verification of an OS Microkernel." In: *ACM Trans. Comput. Syst.* 32.1 (Feb. 2014). ISSN: 0734-2071. DOI: 10.1145/2560537. URL: https://doi.org/10.1145/2560537.

[76] R. Koo and S. Toueg. "Checkpointing and Rollback-Recovery for Distributed Systems." In: *IEEE Transactions on Software Engineering* SE-13.1 (1987), pp. 23–31. DOI: 10.1109/TSE.1987.232562.

[77] Jonathan Koomey. "Growth in data center electricity use 2005 to 2010." In: *A report by Analytical Press, completed at the request of The New York Times* (2011).

[78] K. Kourai and S. Chiba. "A Fast Rejuvenation Technique for Server Consolidation with Virtual Machines." In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 2007, pp. 245–255.

[79] Butler Lampson and Howard E. Sturgis. "Crash Recovery in a Distributed Data Storage System." June 1979. URL: https://www.microsoft.com/en-us/research/publication/crash-recovery-in-a-distributed-data-storage-system/.

[80] Anna Lanzaro, Roberto Natella, Stefan Winter, Domenico Cotroneo, and Neeraj Suri. "An Empirical Study of Injected versus Actual Interface Errors." In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, 397–408. ISBN: 9781450326452. DOI: 10.1145/2610384.2610418. URL: https://doi.org/10.1145/2610384.2610418.

[81] M. Le and Y. Tamir. "Fault Injection in Virtualized Systems – Challenges and Applications." In: *IEEE Transactions on Dependable and Secure Computing* 12.3 (2015), pp. 284–297.

[82] Michael Le, Andrew Gallagher, and Yuval Tamir. "Challenges and opportunities with fault injection in virtualized systems." In: *1st Int. Workshop on Virtualization Performance: Analysis, Characterization, and Tools*. Citeseer. 2008.

[83] Michael Le and Yuval Tamir. "ReHype: Enabling VM Survival across Hypervisor Failures." In: VEE '11. Newport Beach, California, USA: Association for Computing Machinery, 2011, 63–74. ISBN: 9781450306874. DOI: 10.1145/1952682.1952692. URL: https://doi.org/10.1145/1952682.1952692.

[84] Michael Le and Yuval Tamir. *Resilient Virtualized Systems Using ReHype*. 2014. arXiv: 2101.09282 [cs.SE].

[85] Z. Li, Q. Lu, L. Zhu, X. Xu, Y. Liu, and W. Zhang. "An Empirical Study of Cloud API Issues." In: *IEEE Cloud Computing* 5.2 (2018), pp. 58–72. DOI: 10.1109/MCC.2018.022171668.

[86] H. Liu. "A Measurement Study of Server Utilization in Public Clouds." In: *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*. 2011, pp. 435–442.

[87] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. "What Bugs Cause Production Cloud Incidents?" In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '19. Bertinoro, Italy: Association for Computing Machinery, 2019, 155–162. ISBN: 9781450367271. DOI: 10.1145/3317550.3321438. URL: https://doi.org/10.1145/3317550.3321438.

[88] Jiuxing Liu, Wei Huang, Bülent Abali, and Dhabaleswar K Panda. "High Performance VMM-Bypass I/O in Virtual Machines." In: *USENIX Annual Technical Conference, General Track*. 2006, pp. 29–42.

[89] Qinghua Lu, Liming Zhu, Len Bass, Xiwei Xu, Zhanwen Li, and Hiroshi Wada. "Cloud API Issues: An Empirical Study and Impact." In: *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*. QoSA '13. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2013, 23–32. ISBN: 9781450321266. DOI: 10.1145/2465478.2465481. URL: https://doi.org/10.1145/2465478.2465481.

[90] H. Madeira, D. Costa, and M. Vieira. "On the emulation of software faults by software fault injection." In: *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*. 2000, pp. 417–426. DOI: 10.1109/ICDSN.2000.857571.

[91] T. J. McCabe. "A Complexity Measure." In: *IEEE Transactions on Software Engineering* SE-2.4 (Dec. 1976), pp. 308–320. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837.

[92] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. Tech. rep. 800-145. Gaithersburg, MD: National Institute of Standards and Technology (NIST), Sept. 2011. URL: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[93] R. Moraes, R. Barbosa, J. Duraes, N. Mendes, E. Martins, and H. Madeira. "Injection of faults at component interfaces and inside the component code: are they equivalent?" In: *2006 Sixth European Dependable Computing Conference*. 2006, pp. 53–64. DOI: 10.1109/EDCC.2006.16.

[94] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. "The soft error problem: an architectural perspective." In: *11th International Symposium on High-Performance Computer Architecture*. Feb. 2005, pp. 243–247. DOI: 10.1109/HPCA.2005.37.

[95] Shubu Mukherjee. *Architecture design for soft errors*. Morgan Kaufmann, 2008.

[96] P. Musavi, B. Adams, and F. Khomh. "Experience Report: An Empirical Study of API Failures in OpenStack Cloud Environments." In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. 2016, pp. 424–434. DOI: 10.1109/ISSRE.2016.42.

[97] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. "Understanding and Dealing with Operator Mistakes in Internet Services." In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, p. 5.

[98] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira. "On Fault Representativeness of Software Fault Injection." In: *IEEE Transactions on Software Engineering* 39.1 (Jan. 2013), pp. 80–96. ISSN: 0098-5589. DOI: 10.1109/TSE.2011.124.

[99] Michael Nicolaidis. *Soft errors in modern electronic systems*. Vol. 41. Springer Science & Business Media, 2010.

[100] Nitro Enclaves. *Amazon*. https://aws.amazon.com/ec2/nitro/nitro-enclaves/. Accessed: 2020-11-25.

[101] David Oppenheimer, Archana Ganapathi, and David A Patterson. "Why do Internet services fail, and what can be done about it?" In: *USENIX Symposium on Internet Technologies and Systems*. Vol. 67. Seattle, WA. 2003.

[102] David A. Patterson, Garth Gibson, and Randy H. Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)." In: *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*. SIGMOD '88. Chicago, Illinois, USA: Association for Computing Machinery, 1988, 109–116. ISBN: 0897912683. DOI: 10.1145/50202.50214. URL: https://doi.org/10.1145/50202.50214.

[103] Bryan D Payne. *Simplifying virtual machine introspection using LibVMI.* Tech. rep. Sept. 2012. DOI: 10.2172/1055635. URL: https://www.osti.gov/biblio/1055635.

[104] G. Pereira, R. Barbosa, and H. Madeira. "Practical Emulation of Software Defects in Source Code." In: *2016 12th European Dependable Computing Conference (EDCC)*. Sept. 2016, pp. 130–140. DOI: 10.1109/EDCC.2016.19.

[105] Soila Pertet and Priya Narasimhan. *Causes of failure in web applications*. Tech. rep. Technical Report CMU-PDL-05-109, Carnegie Mellon University, 2005.

[106] Jonas Pfoh, Christian Schneider, and Claudia Eckert. "A Formal Model for Virtual Machine Introspection." In: *Proceedings of the 1st ACM Workshop on Virtual Machine Security*. VMSec '09. Chicago, Illinois, USA: Association for Computing Machinery, 2009, 1–10. ISBN: 9781605587806. DOI: 10.1145/1655148.1655150. URL: https://doi.org/10.1145/1655148.1655150.

[107] C. Pham, D. Chen, Z. Kalbarczyk, and R. K. Iyer. "CloudVal: A framework for validation of virtualization environment in cloud infrastructure." In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*. 2011, pp. 189–196.

[108] Riccardo Pinciroli, Lishan Yang, Jacob Alter, and Evgenia Smirni. *The Life and Death of SSDs and HDDs: Similarities, Differences, and Prediction Models*. 2020. arXiv: 2012.12373 [cs.LG].

[109] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. "Failure Trends in a Large Disk Drive Population." In: *5th USENIX Conference on File and Storage Technologies (FAST 2007)*. 2007, pp. 17–29.

[110] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures." In: *Commun. ACM* 17.7 (July 1974), 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073. URL: https://doi.org/10.1145/361011.361073.

[111] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "IRON File Systems." In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. Brighton, United Kingdom: Association for Computing Machinery, 2005, 206–220. ISBN: 1595930795. DOI: 10.1145/1095810.1095830. URL: https://doi.org/10.1145/1095810.1095830.

[112] D. K. Pradhan and N. H. Vaidya. "Roll-forward checkpointing scheme: a novel fault-tolerant architecture." In: *IEEE Transactions on Computers* 43.10 (1994), pp. 1163–1174. DOI: 10.1109/12.324542.

[113] Richard F. Rashid and George G. Robertson. *Accent, a communication oriented network operating system kernel*. June 2018. DOI: 10.1184/R1/6602927.v1. URL: https://kilthub.cmu.edu/articles/journal_contribution/Accent_a_communication_oriented_network_operating_system_kernel/6602927/1.

[114] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. "Towards understanding heterogeneous clouds at scale: Google trace analysis." In: *Intel Science and Technology Center for Cloud Computing, Tech. Rep* 84 (2012).

[115] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. *Google cluster-usage traces: format + schema*. Technical Report. Revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data. Mountain View, CA, USA: Google Inc., Nov. 2011.

[116] G. L. Ries, G. S. Choi, and R. K. Iyer. "Device-level transient fault modeling." In: *Proceedings of IEEE 24th International Symposium on Fault-Tolerant Computing*. June 1994, pp. 86–94. DOI: 10.1109/FTCS.1994.315654.

[117] Ryan Riley, Xuxian Jiang, and Dongyan Xu. "Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing." In: *Recent Advances in Intrusion Detection*. Ed. by Richard Lippmann, Engin Kirda, and Ari Trachtenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–20. ISBN: 978-3-540-87403-4.

[118] Robert Rose. *Survey of system virtualization techniques*. Tech. rep. Oregon State University, 2004.

[119] D. S. Rosenblum. "A practical approach to programming with assertions." In: *IEEE Transactions on Software Engineering* 21.1 (Jan. 1995), pp. 19–31. ISSN: 0098-5589. DOI: 10.1109/32.341844.

[120] M. Rosenblum and T. Garfinkel. "Virtual machine monitors: current technology and future trends." In: *Computer* 38.5 (2005), pp. 39–47. DOI: 10.1109/MC.2005.176.

[121] B. Sangchoolie, K. Pattabiraman, and J. Karlsson. "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)*. Vol. 00. June 2017, pp. 97–108. DOI: 10.1109/DSN.2017.30.

[122] B. Schroeder and G. A. Gibson. "A Large-Scale Study of Failures in High-Performance Computing Systems." In: *IEEE Transactions on Dependable and Secure Computing* 7.4 (2010), pp. 337–350. DOI: 10.1109/TDSC.2009.4.

[123] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. "Flash Reliability in Production: The Expected and the Unexpected." In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 67–80. ISBN: 978-1-931971-28-7.

[124] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. "DRAM Errors in the Wild: A Large-scale Field Study." In: *SIGMETRICS Perform. Eval. Rev.* 37.1 (June 2009), pp. 193–204. ISSN: 0163-5999. DOI: 10.1145/2492101.1555372.

[125] Thomas Schwarz, Mary Baker, Steven Bassi, Bruce Baumgart, Catherine Van Ingen, Kobus Joste, Mark Manasse, and Mehul Shah. "Disk failure investigations at the internet archive." In: *NASA/IEEE Conference on Mass Storage Systems and Technologies (MSST)*. 2006.

[126] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Ysskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin. "FIAT - Fault injection based automated testing environment." In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'*. June 1995, pp. 394–. DOI: 10.1109/FTCSH.1995.532663.

[127] Arman Shehabi, Sarah Smith, Dale Sartor, Richard Brown, Magnus Herrlin, Jonathan Koomey, Eric Masanet, Nathaniel Horner, Inês Azevedo, and William Lintner. *United States Data Center Energy Usage Report*. Tech. rep. June 2016. DOI: 10.2172/1372902. URL: https://www.osti.gov/biblio/1372902.

[128] S. Shen, V. Van Beek, and A. Iosup. "Statistical Characterization of Business-Critical Workloads Hosted in Cloud Datacenters." In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2015, pp. 465–474. DOI: 10.1109/CCGrid.2015.60.

[129] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. *RFC3010: NFS Version 4 Protocol*. USA, 2000.

[130] Shielded VMs | Google Cloud. *Google*. https://cloud.google.com/shielded-vm. Accessed: 2020-11-25.

[131] D. Skarin, R. Barbosa, and J. Karlsson. "GOOFI-2: A tool for experimental dependability assessment." In: *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*. June 2010, pp. 557–562. DOI: 10.1109/DSN.2010.5544265.

[132] David Smiley, Eric Pugh, Kranti Parisa, and Matt Mitchell. *Apache Solr enterprise search server*. Packt Publishing Ltd, 2015.

[133] J. E. Smith and Ravi Nair. "The architecture of virtual machines." In: *Computer* 38.5 (2005), pp. 32–38. DOI: 10.1109/MC.2005.173.

[134] Ian Sommerville. *Software Engineering*. 9th. USA: Addison-Wesley Publishing Company, 2010. ISBN: 0137035152.

[135] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly." In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: Association for Computing Machinery, 2015, 297–310. ISBN: 9781450328357.

DOI: 10.1145/2694344.2694348. URL: https://doi.org/10.1145/2694344.2694348.

[136] Neil R. Storey. *Safety Critical Computer Systems*. USA: Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN: 0201427877.

[137] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. K. Iyer. "NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors." In: *Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000*. 2000, pp. 91–100. DOI: 10.1109/IPDS.2000.839467.

[138] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. "Recovering Device Drivers." In: *ACM Trans. Comput. Syst.* 24.4 (Nov. 2006), 333–360. ISSN: 0734-2071. DOI: 10.1145/1189256.1189257. URL: https://doi.org/10.1145/1189256.1189257.

[139] TPC. *TPC Virtual Measurement Single System (TPC-VMS), Version 1.2.0*. 2013.

[140] Cheng Tan, Yubin Xia, Haibo Chen, and Binyu Zang. "TinyChecker: Transparent protection of VMs against hypervisor failures with nested virtualization." In: *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*. June 2012, pp. 1–6. DOI: 10.1109/DSNW.2012.6264691.

[141] Tcpdump. *TCPDump*. https://www.tcpdump.org/. Accessed: 2021-02-01. 2019.

[142] AWS Team. *Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region*. https://aws.amazon.com/message/65648/. Accessed: 2021-01-01. 2011.

[143] AWS Team. *Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region*. https://aws.amazon.com/message/680587/. Accessed: 2021-01-01. 2012.

[144] AWS Team. *Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region*. https://aws.amazon.com/message/41926/. Accessed: 2021-01-01. 2017.

[145] Anna Thomas and Karthik Pattabiraman. "LLFI: An intermediate code level fault injector for soft computing applications." In: *Workshop on Silicon Errors in Logic System Effects (SELSE)*. 2013.

[146] Timothy K. Tsai and Ravishankar K. Iyer. "Measuring Fault Tolerance with the FTAPE fault injection tool." In: *Quantitative Evaluation of Computing and Communication Systems*. Ed. by Heinz Beilner and Falko Bause. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 26–40. ISBN: 978-3-540-44789-4.

[147] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. "Intel virtualization technology." In: *Computer* 38.5 (2005), pp. 48–56. DOI: 10.1109/MC.2005.163.

[148] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. "Characterizing Cloud Computing Hardware Reliability." In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, 193–204. ISBN: 9781450300360. DOI: 10.1145/1807128.1807161. URL: https://doi.org/10.1145/1807128.1807161.

[149] Cheng Wang, Xusheng Chen, Weiwei Jia, Boxuan Li, Haoran Qiu, Shixiong Zhao, and Heming Cui. "PLOVER: Fast, Multi-core Scalable Virtual Machine Fault-tolerance." In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 483–489. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/wang.

[150] Wei-Lun Kao and R. K. Iyer. "DEFINE: a distributed fault injection and monitoring environment." In: *Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*. 1994, pp. 252–259. DOI: 10.1109/FTPDS.1994.494497.

[151] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble. "Rethinking the design of virtual machine monitors." In: *Computer* 38.5 (2005), pp. 57–62. DOI: 10.1109/MC.2005.169.

[152] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. "Scale and Performance in the Denali Isolation Kernel." In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2003), 195–209. ISSN: 0163-5980. DOI: 10.1145/844128.844147. URL: https://doi.org/10.1145/844128.844147.

[153] Wikimedia. *enwiki dump progress on 20200301*. https://dumps.wikimedia.org/enwiki/20200301/. Accessed: 2020-04-20.

[154] John Wilkes. *More Google cluster data*. Google research blog. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html. Mountain View, CA, USA, Nov. 2011.

[155] X. Xu and H. H. Huang. "DualVisor: Redundant Hypervisor Execution for Achieving Hardware Error Resilience in Datacenters." In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. May 2015, pp. 485–494. DOI: 10.1109/CCGrid.2015.30.

[156] X. Xu and H. H. Huang. "On Soft Error Reliability of Virtualization Infrastructure." In: *IEEE Transactions on Computers* 65.12 (2016), pp. 3727–3739.

[157] Xin Xu and H Howie Huang. “Understanding reliability implication of hardware error in virtualization infrastructure.” In: *10th Workshop on Hot Topics in System Dependability (HotDep 14)*. 2014.

[158] Wei Zheng, Chen Feng, Tingting Yu, Xibing Yang, and Xiaoxue Wu. “Towards understanding bugs in an open source cloud management stack: An empirical study of OpenStack software bugs.” In: *Journal of Systems and Software* 151 (2019), pp. 210 –223. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.02.025. URL: http://www.sciencedirect.com/science/article/pii/S0164121219300354.

[159] D. Zhou and Y. Tamir. “Fast Hypervisor Recovery Without Reboot.” In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2018, pp. 115–126.

[160] jTPCC developers. *jTPCC*. http://jtpcc.sourceforge.net/. Accessed: 2017-02-05.

[161] E. van der Kouwe, C. Giuffriday, R. Ghituletez, and A. S. Tanenbaum. “A Methodology to Efficiently Compare Operating System Stability.” In: *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*. 2015, pp. 93–100. DOI: 10.1109/HASE.2015.22.