UMLinux - A Tool for Testing a Linux System’s Fault Tolerance

Hans-Jörg Höxer

Universität Erlangen-Nürnberg
Institut für Informatik 3

Martensstr. 3 91058 Erlangen Germany
hshoexer@immd3.informatik.uni-erlangen.de

Kerstin Buchacker

Universität Erlangen-Nürnberg
Institut für Informatik 3

Martensstr. 3 91058 Erlangen Germany
kerstin.buchacker@immd3.informatik.uni-erlangen.de

Volkmar Sieh

Universität Erlangen-Nürnberg
Institut für Informatik 3

Martensstr. 3 91058 Erlangen Germany
volkmar.sieh@immd3.informatik.uni-erlangen.de

Abstract

When setting up servers it would often be nice to know, how these systems will react to hardware-failures such as a defect harddisk, random access memory, network interface or simple power failure. Will data be lost or corrupted or will the system simply not be accessible for clients for some time? The silent corruption of data without any error messages, for example, is a worst case scenario for database systems.

It would be nice to be able to test, if a system designed to continue delivering services even in the presence of faults will indeed do so.

We have implemented UMLinux, a User Mode Linux which can be used for realistic fault injection experiments to help answer the above questions. In order to be as close to reality as possible, our UMLinux implements kernel memory protection and runs the complete virtual machine (including operating system and all processes) as a single process on the (real) host. Of course, UMLinux is binary compatible with the host, so all binaries which run on the host also run on UMLinux (without recompilation).

The system we want to examine is set up using virtual UMLinux machines. For most Linux-distributions we can use the out-of-the-box installation routine to install the virtual machine directly from cdrom. The virtual hardware, such as random access memory size, hard-, floppy- and cdrom-drives as well as network-interfaces can be configured freely within the limits posed by the resources available on the host.

When the virtual server system is up and running, the fault injector can be configured to inject faults into the virtual hardware. We can currently inject bitflips into CPU-registers and main memory, defect bytes on any kind of block device and network send and receive failures.

The whole setup is currently controlled via a graphical user frontend. We are working to implement a script-driven automatic experiment controller.

1. Introduction

This paper presents UMLinux, a tool for testing the fault tolerance behavior of networked machines running the Linux operating system. UMLinux is in some ways similar but by no means identical to Jeff Dike's User Mode Linux UML [Dike01].

The tool simulates a system of Linux machines. The simulated hardware can be made to fail and the reaction of Linux and/or applications running on top of Linux can be observed. The simulation environment is made available by porting the Linux operating system to a new "hardware" — the Linux operating system! The basic principles are similar to those known from Jeff Dike's User Mode Linux UML [Dike01]. Due to the *binary compatibility* of the simulated and the host system, any program that runs on the host system will also run on the simulated machine.

A tracer process paired with each simulated machine injects faults via the `ptrace` interface. This interface allows complete control over the simulator process, including access to registers and memory as well as to arguments and return values of input/output operations. Thus, the virtual hardware can be made to fail in a number of possible ways, including hardware faults in computing core and peripheral devices of a single (virtual) machine as well as faults external to machines, such as faults in (virtual) external networking hardware.

Because of its fault injection capabilities and the fact that it is software-based (no faults are injected into any real hardware), UMLinux can count among the software implemented fault injection (SWIFI) tools, such as MEFISTO [JARO94], VERIFY [SiTB97], CrashMe [Carr96], Fuzz [MKLM95], FERRARI

[KaKA92], MAFALDA [RSFA93], a fault-injector based on the `ptrace` interface [Sieh93], FIAT [BCSS90], Xception [CaMS95], and Ballista [KrKS98], all of which are presented in a little more detail in [BuSi01]. UMLinux has a number of advantages over traditional SWIFI tools which are explained in [BuSi01].

The tool will be used in the European DBench Project [dben01] for dependability benchmarking of Linux systems.

For information about the implementation of UMLinux please refer to [BuSi01].

1.1. Current and Planned Capabilities

Current capabilities of a single virtual machine include running a number of different out-of-the-box Linux distributions in a non-graphical mode. X support is currently very basic since we do not yet have mouse support. The framebuffer X server is able to run locally. The virtual machines all have full networking capabilities. Faults can be injected into the virtual hardware and information from the fault injector can be logged to a file. A prototype experiment controller exists which can simulate keyboard input and evaluate the console output of a virtual machine.

We plan to implement full mouse and X support in the near future. This will include a virtual graphics card with configurable on-board graphics memory. Another ongoing project is the separation of virtual hardware and kernel parts with the goal to use as much of the original kernel as possible. Our first approach was simply to add new drivers for our virtual hardware to the original kernel. Since we want to use the original drivers, we will need to implement something that acts like the hardware the original driver was written for. We have already done this with the keyboard controller and are currently working on accomplishing the same thing with the IDE subsystem. It is possible that some of the virtual hardware (such as onboard chipsets/controllers) will in fact run as separate processes. This is a good model for controllers, since the Linux driver simply sends commands to the controller and expects/evaluates the answers. To be able to use UMLinux for testing and fault injection experiments we need an experiment controller which allows full automization of a testrun. The experiment controller should be able to send any kind of "commands from the environment", such as characters typed at the keyboard, mouse movements, cdrom-eject-button push etc. to the virtual machine which should then react accordingly. The virtual machine should be able to send all kinds of logging output to the experiment controller for evalutation. This kind of logging output could include console output (this is already implemented) or information about menus or highlighted parts in X as well as information from the tracer/fault injector.

2. Differences between UMLinux and Jeff Dike's UML

At first sight, the UML described in [Dike01] has a number of similarities with the user mode port of Linux we present in this paper. When taking a closer look, major differences become clear.

2.1. User Mode Processes

One major difference is, that in [Dike01] a design decision was made, to map *every* UML process onto a *separate* process in the host system. From the latter follows, that parts of the information any operating system must keep for each process will not be kept by the UML kernel, but by the kernel of the host system. Of course, information kept by the host kernel will not be affected by faults injected into the virtual main memory of the UML kernel, with the result, that injected memory faults cannot affect all processes (as would be the case in a real world system). In our UMLinux implementation the simulated machine (including its operating system and all the processes running on it) is therefore implemented as a *single* real process.

2.2. Kernel Memory Protection

Another big difference is, that UML [Dike01] does not yet implement kernel memory protection. This will cause differences in behavior to a real Linux kernel, when user processes write into the kernel memory space maliciously or due to an injected fault. In a real Linux kernel, illegal access of kernel memory space by a user process will usually crash the user process, whereas the kernel memory remains intact. In UML — because of the missing kernel memory protection — the kernel memory will also be corrupted. Therefore, to make UMLinux behave like a real Linux, we had to implement kernel memory protection.

2.3. Console and X

A virtual machine running under UMLinux uses a low level console-driver similar to the other console drivers found in `linux/drivers/video/`. All driver output is sent to the frontend, which acts like a console and displays a special console window for each virtual machine. UML on the other hand [Dike01] uses **xterm** as default console for the virtual machine (a number of other different terminals can be configured). The termios functions are used for transmitting data to the terminal.

UMLinux will be using an adapted framebuffer device to run the **XF86_FBDev** X server locally on the virtual machine. The implementation is already running for one kernel and we are currently working on integrating it into the other kernels. Mouse support is still missing but is planned for a future release. UML runs **XNest**, the nested X server available from The XFree86 Project, Inc. (http://www.xfree86.org/), locally as a server on the virtual machine. **XNest** behaves like a client to the X server on the real machine, so for this setup to work, the real and virtual machines must be connected via a (virtual) network. Of course the same setup can also be used with UMLinux.

3. Fault Injection

This section explains, how the fault injection of hardware faults is implemented in UMLinux. We do not have a closer look at system configuration faults, as these are trivially injected by copying the faulty configuration files onto the system.

To inject faults into the virtual hardware, those parts of the simulator implementing the hardware must be accessed. This can be achieved via the `ptrace` interface. All system calls can of course be modified to implement fault injection, but those system calls implementing the UMLinux hardware are most important for injecting (UMLinux) hardware faults.

To minimize the overhead, the tracer intercepting and diverting the system calls, also handles the fault injection.

The tracer reads the faults to be injected from a file when the simulator starts. A configured fault becomes active at the time given in the file. Once one or more faults are active, the appropriate actions must be taken to inject the fault.

3.1. Computing Core Faults.

Memory faults include transient bit-flip and permanent stuck-at faults. It is no problem to inject transient faults into the virtual machine. This is done by simply writing to the memory mapped file which is the virtual machine's RAM. Transient faults can only affect processes (including the UM kernel) on the virtual machine when they read from memory, but will be overwritten by write accesses to the faulty part. Permanent faults, which do not disappear after a write access to the faulty part, can be implemented by remapping the affected pages or (on up-to-date Intel hardware) using the user debugging registers provided by Intel processors.

CPU faults include transient bit flips or permanent stuck-at faults in registers. An effect may be instructions skipped or wrong branches taken. Again, injecting transient faults is no problem, since a full copy of the register contents is passed to the tracer via the `ptrace` interface and modified register contents can be passed back to the simulator. To implement permanent faults, the register contents have to be checked and possibly modified after every single instruction executed by the simulator, which will lead to a higher overhead. This can be accomplished by single stepping the simulator.

Faults injected into the computing core will affect both the UM kernel and all UM user processes on the given virtual machine, just as would be the case on a real machine.

3.2. Peripheral Faults.

Peripheral hardware access, such as harddisk, floppy or cdrom drive access is implemented using the `open`, `close`, `read`, `write` and `lseek` system calls. Thus the arguments of these system calls are checked to see if the faulty device is being accessed. If it is, the return value from a `read` or the data passed to a `write` is modified according to the fault definition, for example to implement several defect blocks on a harddisk. An inaccessible harddisk is implemented by modifying the return value of all above system calls to return an error.

To inject faults into the network interface, the arguments and return values of the appropriate system calls are modified. By modifying the return value of a `recvfrom`, for example, IP packets with faulty protocol information or wrong checksums can be generated.

Injecting permanent faults into peripheral devices does not incur such a high overhead as injecting permanent faults into the computing core, since the simulator is stopped at every system call anyway, so that the tracer can redirect the system call to the UM kernel if necessary. Additionally, only those system calls implementing UMLinux drivers need to be examined more closely when peripheral faults are active. Those system calls redirected into the UMLinux kernel need not be manipulated to inject peripheral faults.

3.3. External Faults.

Depending on the setup, a power failure may affect a single or several machines. For all virtual machines affected by the virtual power failure, the tracer simply kills the corresponding processes by sending them a `SIGKILL`.

Defects in virtual external networking hardware can be implemented by configuring the UM networking process to behave like faulty networking hardware. This is a separate process which transparently passes packets between virtual and real machines and thus integrates the virtual network into the real network. By configuring the UM networking process to not forward packets to/from a certain machine, a broken cable leading to that machine can be simulated. Examples for other possible defects include a working uplink but a broken downlink or missing or damaged network packets.

3.4. Interrupt/Exception Faults.

The tracer can intercept signals before they are delivered to the virtual machine as interrupts or exceptions. It is thus possible to inject "missing interrupt" faults, where an exception or an interrupt is not generated even though this should have been the case.

4. Experiments

This sections describes a simple example experiment conducted using UMLinux. The general setup of the example system is the following:

System under test:

`DNS`: domain name-server running bind.

`DB_WWW`: web and database-server running Apache and MySQL. This virtual machine was equipped with two harddisks, one containing the operating system and binaries (`HD1`), the other containing the database and HTML-pages for the webserver (HD2). The contents of the database were generated automatically and consist of timestamped records each with a primary key. Two different databases were used. One contained about 2.7 million entries (DB1), the other started out empty and was filled and emptied again during the testrun (DB2).

Network:

The virtual machines were connected by a virtual local network.

Processor, memory, harddisk and network faults were injected into DB_WWW.

The workload was generated by Perl-scripts running on an additional virtual machine (CLIENT). Two different workloads were used.

WL1

230 SELECT statements made via the webinterface accessing records evenly distributed throughout the database.

WL2

A series of 100 INSERT, followed by 150 SELECT and 100 DELETE statements (all randomly generated and submitted via the webinterface) was repeated twice on different record sets.

The record sets to be read and written by the client were prepared in advance and known to the client, such that the client was able to recognize a faulty record returned by the server.

Four different experiments were conducted, one for each type of fault, as described in the following list. The results are summarized in the next paragraphs. The results were extracted from client logfiles.

Memory Faults:

The faultload was a single transient bitflip of a randomly chosen bit in a randomly chosen byte of memory between 0 and 32MB. An equal percentage of runs was made with activation times of 150, 200 and 250 seconds. The workload and database used were WL2 and DB2. 282 single runs were conducted.

Processor Faults:

The faultload was a single transient bitflip of a randomly chosen bit in a randomly chosen register. An equal percentage of runs was made with activation times of 150, 200 or 250 seconds. The workload and database used were WL2 and DB2. 447 single runs were conducted.

Harddisk Faults:

The faultload consisted of permanent failures of 2000 consecutive blocks on the harddisk (HD2) containing the data. The activation time was 200, the start block was randomly chosen on the harddisk. The workload and database used were WL1 and DB1. 726 single runs were conducted.

Network Faults:

> The faultload consisted of transient failures of the network device of `DB_WWW`. Both send and receive failures with durations from 5 to 40 seconds (with step of 5) were injected, with activation time being 150 seconds. The workload and database used were `WL2` and `DB2`. 109 single runs were conducted.

The server’s behavior was viewed from the client’s point of view and the errors observed were therefore classified into the following categories

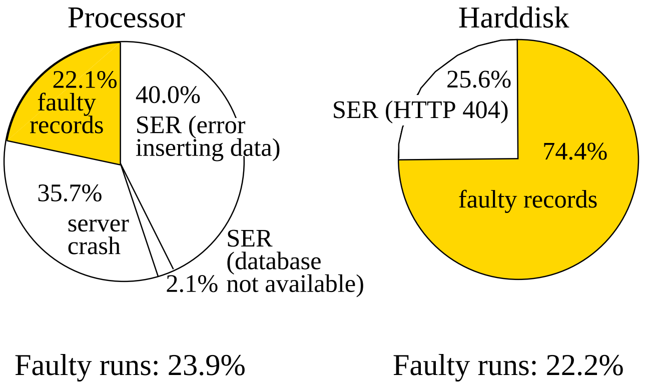
- faulty response (faulty record data)
- delayed response
- server error response (SER)
- server crash or hang

The item SER corresponds to the HTTP-server returning some kind of error message, such as a "page not found" message or error messages from the database server which are passed on to the client via the HTTP-server. The last item is a server crash or hang from the clients point of view, i.e. the client is unable to evoke a response from the server until the end of the testrun. Not all of these possible behaviors were observed for each type of fault injected.

The memory faults injected had no immediately visible effect on the server. We believe this is due to the fact that only a single fault was injected per testrun. We also did not try to target sensitive parts of the memory explicitly, since (apart from the memory location of the kernel) it is not possible to tell a priori where i.e. the database- or webserver executables are located in memory at a certain time during the testrun. This behavior has also been observed on real machines with a defect RAM, were the only visible errors occurring once in a while were some defect files on the harddisk (the reason for this being the fact that Linux buffers disk I/O, so a memory fault in one of the I/O buffers will affect what is written to disk).

Figure 1 shows the percentage of different types of behavior observed in the example setup for the processor and harddisk faults.

Figure 1. Results of the Experiments (Faulty Runs Only)



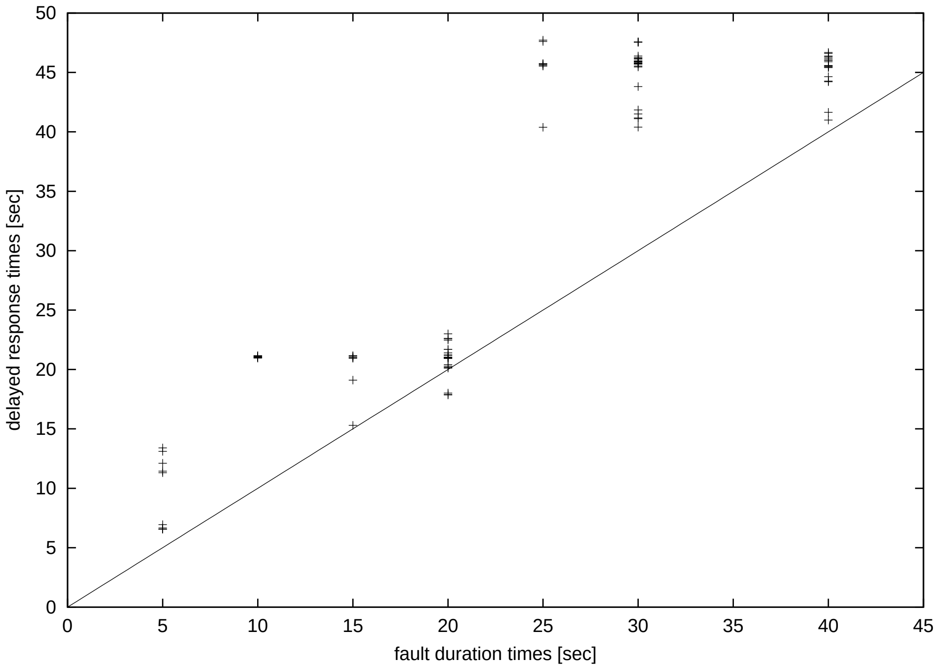
For 86.1% of the testruns injecting processor faults, the client could not observe a faulty server behavior. For the 23.9% of testruns with observable faulty behavior, the distribution is shown in the left part of Figure 1. It is possible for several different errors to occur during a single testrun. The client completely lost connection with the server and could not regain it during this testrun (35.7% of the faults). For the client it is impossible to tell, whether the lost connection is due to an operating system crash of the server or crash of the webserver daemon only. In 40.0% of the errors observed, the server returned an error message, saying, that it could not insert the data into the database. In 22.1% of the cases no error message but a faulty record were returned. In the last 2.1% the web server returned the message, that it was unable to connect to the database.

Harddisk faults, since confined to `HD2`, could not affect the operating systems or database- and webserver binaries. Accordingly the clients never lost connection to the webserver, instead the webserver returned error messages when the data the client requested was inaccessible. Of the testruns performed, 77.8% terminated without errors. The percentages of the errors observed in the other 22.2% of the testruns are shown in the right part of Figure 1. In about a quarter of the cases the web server returned an HTTP 404 "page not found" error, the rest of the time faulty records were returned without any error messages.

The experiment shows, that the worst case, i.e. undetectable errors, happens in, as we believe, a non-negligible percentage of the testruns. In the experiment setup, the clients knew which response to expect from the server and could therefore identify the faulty records returned. This is usually not the case in a real world system.

Network faults only led to delayed server responses being observed by the clients. This is due to the fact, that the HTTP-exchange between client and server is layered on the fault-tolerant Transmission Control Protocol (TCP). The latter hides the retransmissions occurring due to the network failure from the application layer and the client only records a higher response time. The durations of the network faults were obviously not long enough to lead to TCP-timeouts. Figure 2 shows how the response times of the delayed responses (y-axis) relate to the fault duration (x-axis). The response times are sometimes much higher than the actual fault duration. This is due to the backoff and retry mechanism of TCP, which backs off for an increasing amount of time after an unsuccessful retry before trying again to connect.

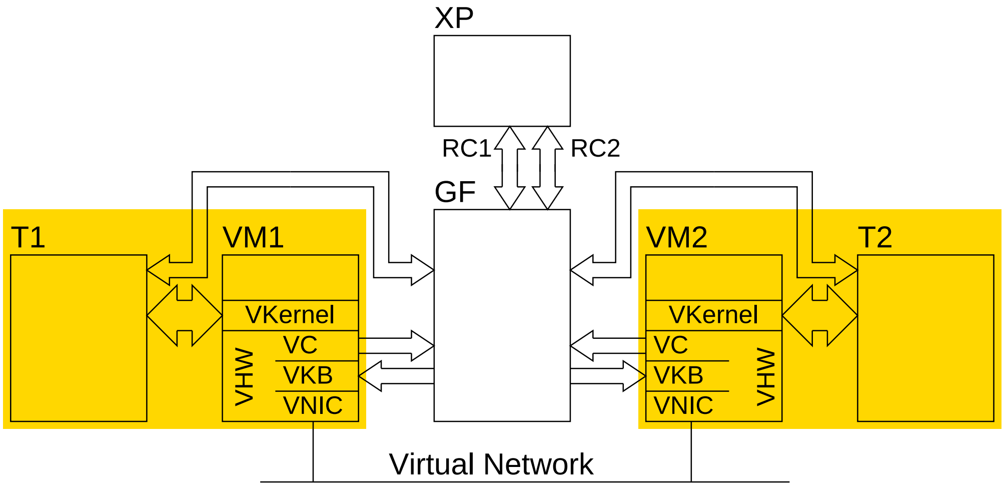
Figure 2. Network Delay Times



5. Demonstration

The tool consists of several interacting processes. An abstract view of the tool is shown in Figure 3.

Figure 3. Tool Overview



The figure shows the processes of the tool when two virtual machines (VM1 and VM2) are started. Each virtual machine is paired with its tracer (T), which is also the fault injector. Every box is a single process on the host. The processes making up a VM-T pair (shaded background) must run on the same host, but other than that there is no restriction. If several physical hosts are available, it is convenient to balance the load by starting VM-T pairs on different hosts. The close interaction of the tracer and the virtual machine is symbolized in the figure with a wide arrow.

The narrow arrows show how the graphical frontend (GF) interacts with the virtual machines. The output of each virtual console (VC) is sent to the frontend, where it can be viewed or saved. The frontend can simulate users sitting in front of the virtual machines typing away at the keyboard, since the virtual keyboard (VKB) is taking input from the frontend. The frontend must be started to use the UMLinux. The frontend can switch between the virtual machines, so a single instance is sufficient to control a complete virtual system under test consisting of several virtual machines.

When the frontend is started by the automatic experiment controller prototype (XP), it opens a remote control connection for each virtual machine (RC1, RC2). The frontend then passes all console output received by the virtual machines to the experiment controller, which may return keyboard input as appropriate.

The demonstration will show the frontend controlling two UMLinux machines which are connected to the same virtual network. We will show how UMLinux reacts to a failure of the virtual cdrom while reading from this device by injecting cdrom-failures during a copy from cdrom to harddisk. With a simple **ping** setup of two virtual machines each **ping**ing the other, network faults such as send- or receive failures can be made visible.

References

[BCSS90] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek, “Fault Injection Experiments Using FIAT”, 1990, 39, 4, 575–582, *IEEE Transactions on Computers*.

[BuSi01] Kerstin Buchacker and Volkmar Sieh, “Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects”, 2001, 95–105, *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium*.

[CaMS95] J. Carreira, H. Madeira, and J. G. Silva, “Xception: Software Fault Injection and Monitoring in Processor Functional Units”, 1995, 135-149, *5th International Working Conference on Dependable Computing for Critical Applications*.

[Carr96] G. J. Carrette, *CrashMe*, 1996.

[dben01] DBench - Dependability Benchmarking (Project IST-2000-25425), *Coordinator: Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique, Toulouse, France; Partners: Chalmers University of Technology, Göteborg, Sweden; Critical Software, Coimbra, Portugal; Faculdade de Ciencias e Technologia da Universidade de Coimbra, Portugal; Friedrich-Alexander Universität, Erlangen-Nürnberg, Germany; Microsoft Research, Cambridge, UK; Universidad Politechnica de Valencia, Spain*, 2001.

[Dike01] J. Dike, “A User-Mode Port of the Linux Kernel”, 2001, *5th Annual Linux Showcase & Conference, Oakland, California*.

[JARO94] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, “Fault Injection into VHDL Models: The MEFISTO Tool”, 66–75, 1994, *Proceedings of the 24th IEEE International Symposium on Fault Tolerant Computing*.

[KaKA92] G. Kanawati, N. Kanawati, and J. Abraham, “FERRARI: A Tool for the Validation of System Dependability Properties”, 1992, 336–344, *Proceedings of the 22th IEEE International Symposium on Fault Tolerant Computing*.

[KrKS98] N. Kropp, P. J. Koopman, and D. P. Siewiorek, “Automated Robustness Testing of Off-the-Shelf Software Components”, 1998, 230–239, *Proceedings of the 28th IEEE International Symposium on Fault Tolerant Computing*.

[MKLM95] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, *Fuzz Revised: A Re-examination of the Reliability of UNIX Utilities and Services*, University of Wisconsin-Madison, 1995, Computer Science Technical Report 1268.

[RSFA93] M. Rodríguez, F. Salles, J. C. Fabre, and J. Arlat, “MAFALDA: Microkernel Assessment by Fault Injection and Design Aid”, 1993, 208–217, *3rd European Dependable Computing Conference*.

[Sieh93] Volkmar Sieh, *Fault-Injector Using UNIX ptrace Interface*, IMMD3, Universität Erlangen-Nürnberg, 1993, Internal Report 11/93.

[SiTB97] Volkmar Sieh, Oliver Tschäche, and Frank Balbach, “VERIFY: Evaluation of Reliability Using VHDL-Models with Integrated Fault Descriptions”, 32–36, 1997, *Proceedings of the 27th IEEE International Symposium on Fault Tolerant Computing*.

[SiBu02] Volkmar Sieh and Kerstin Buchacker, “Testing the Fault-Tolerance of Networked Systems ”, 2002, 95–105, *International Conference on Architecture of Computing Systems ARCS 2002, Workshop Proceedings* .

[SPEC00] Standard Performance Evaluation Corporation, *SPECweb99 Release 1.02* , 2000.

[TPC01] Transaction Processing Performance Council, *TPC Benchmark [tm] C, Standard Specification, Revision 5.0, February 26, 2001* , 2001.

[Zhan00] Wensong Zhang, “Linux Virtual Server for Scalable Network Services ”, 2000, , *Ottawa Linux Symposium 2000* .