Title: **UMLinux - A Versatile SWIFI Tool**

Authors: Volkmar **Sieh**, Kerstin **Buchacker**

Published In: Proceedings of the Fourth European Dependable Computing Conference, Toulouse, France, October 23-25, 2002

Year: 2002

Pages:

UMLinux — A Versatile SWIFI Tool

Volkmar Sieh and Kerstin Buchacker

Institut für Informatik 3
Friedrich Alexander Universität Erlangen-Nürnberg
Germany
{volkmar.sieh, kerstin.buchacker}@informatik.uni-erlangen.de

Abstract. This tool presentation describes UMLinux, a versatile framework for testing the behavior of networked machines running the Linux operating system in the presence of faults. UMLinux can inject a variety of faults into the hardware of simulated machines, such as faults in the computing core or peripheral devices of a machine or faults in the network connecting the machines. The system under test, which may include several machines, as well as the fault- and workload run on this system are configurable.
UMLinux has a number of advantages over traditional SWIFI and simulation tools: speed, immunity of fault-injection and logging processes from the state of the machine into which the faults are injected and binary compatibility with real world data and programs.

1 Introduction

This tool presentation describes UMLinux, a framework capable of evaluating the dependability behavior of networked machines running the Linux operating system in the presence of faults. The Linux operating system is usually employed in networked server environments, for example as web- or mailserver.

The tool uses software implemented fault injection (SWIFI) to inject faults into a simulated system of Linux machines. The simulation environment is made available by virtualization, i.e. by porting the Linux operating system to a new ”hardware” — the Linux operating system! 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 process paired with each simulated machine injects faults via the `ptrace` interface. This interface allows complete control over the traced process, including access to registers and memory as well as to arguments and return values of input/output operations. Possible faults include hardware faults in computing core and peripheral devices of a single machine as well as faults external to machines, such as faults in external networking hardware.

The tool will be used in the European DBench Project [6] for dependability benchmarking of Linux systems.

The rest of the paper is structured as follows. Section 2 gives a short overview of the main advantages of UMLinux over traditional SWIFI and simulation tools. Section 3 gives an outline of the different parts of the tool. Information about the configuration of

the simulated hardware is found in Sect. 4. Section 5 explains how to inject faults using UMLinux. Section 6 describes an example experiment. The final section outlines a tool demonstration. For information about the implementation of UMLinux please refer to [2].

2 Advantages of UMLinux

Since the issues and problems in implementing a software injection tool at the operating system level as well as the technical details of UMLinux have been treated in [2, 8, 17] this tool presentation will concentrate on the user perspective of UMLinux. UMLinux has advantages over pure simulation and SWIFI tools or virtualization software, because it combines all three — simulation, SWIFI and virtualization. To our knowledge, there is currently no other such tool available worldwide.

[2] gives a short overview of a number of available simulation and SWIFI tools, including VHDL-based simulation ([9, 18]), CrashMe [4], Fuzz [13], FERRARI [10], MAFALDA [14], a fault-injector based on the `ptrace` interface [16], FIAT [1], Xception [3], and Ballista [11].

UMLinux differs from the SWIFI and simulation work named in the previous paragraph in several important aspects. It *combines* SWIFI with a simulation approach and therefore offers all the possibilities of the former. Since we simulate the hardware at a relatively high level, the simulation is unusually fast (slowdown is less then one order of magnitude). Using SWIFI together with a simulated machine has the advantage, that once set up, no user interaction is required.

When using a SWIFI tool to inject faults into the operating system of the machine the tool is actually running on, the faults may also affect the integrity of the tool and cause erroneous results to be logged. In addition, automating testing is difficult, since the machine must usually be rebooted manually when the operating system crashes or hangs and test results may be lost. In UMLinux, on the other hand, the faults are injected into a simulated machine and the fault injection software is not in any way dependent on the integrity of this simulated machine. The integrity of the host machine is in no way affected by the faults injected into the simulated machine. Since the fault injection code is separated from the code for the simulated machine and runs as a separate process, undesired interference and intrusion of the fault injection code on the simulated machine is avoided. Additional capabilities *not* offered by other simulation or SWIFI techniques include fault injection into a system of networked machines.

When using a simulation to represent a real world system, the most important question is, how closely does the simulation mimic the behaviour of the real world system? And inversely, how closely will the real world system follow the behaviour of the simulation in the presence of faults? UMLinux machines can run unmodified real world binaries and can directly use disk images of real world machines. This means that almost the complete software and data part of a UMLinux machine is identical to that of a real world machine. The differences are in the hardware and the closely hardware related software parts (drivers). These are the same differences you encounter when you install the same software onto two machines with slightly different hardware (e.g. harddisks, cdroms, motherboard from different manufacturers). The difference in behaviour

between UMLinux and a real world machine is therefore no greater than the difference in behaviour between two real world machines with slightly different hardware.

In this and the following sections, the term *host* will always be used to designate the physical machine including its operating system. The terms *virtual* or *user mode* (*UM*) will be used interchangeably when the simulated machine and operating system or processes running on this simulated machine are being referred to. [2] lists a number of available virtualization tools, including Bochs [19], SimOS [15], Simics™[20], Plex86 [12], VMware™[21], Virtual PC™[5], User Mode Linux (UML) [7].

To be able to inject a variety of faults precisely at the targeted fault locations, it is necessary to know in detail, how the simulated hardware is implemented. For the commercial products Simics™, VMware™ and Virtual PC™ the source code is not available to us. Therefore we decided not to use them as the core of our fault injection tool. For performance reasons we also did not want to use the tools with processor emulation (Bochs, SimOS). We did not consider using Plex86 as it is still in an experimental phase.

At first sight, the UML described in [7] 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. The first big difference is, that UML [7] does not yet implement kernel memory protection. This will cause differences in behaviour 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 have implemented kernel memory protection. Another major difference is, that in [7] 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.

3 Overview of UMLinux

The tool consists of several interacting processes. An abstract view of the tool is shown in Fig. 1. This tool presentation describes UMLinux from a user's perspective. The technical and implementation details have already been presented in [2, 8, 17]. 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

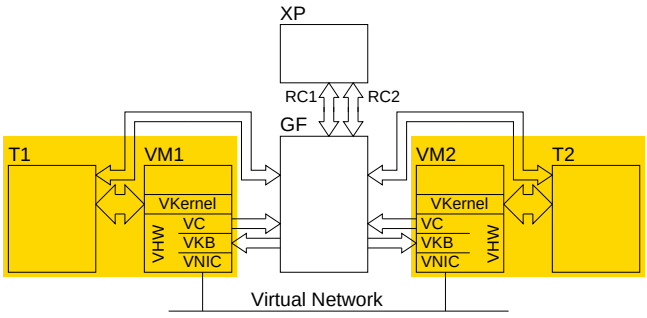

Fig. 1. Tool overview

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, saved or fed into Expect (XP), an automatic control program, as necessary. The frontend (or Expect) 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, whereas Expect is optional and need only be started when running prepared experiments without human interaction. 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. Expect opens a remote control connection (RC) to the frontend for every virtual machine to be controlled. Expect is configured from a file.

Fig. 2 shows a screenshot of the graphical user frontend. Two machines have been started. The window in the middle shows a UMLinux machine which is just booting. Due to a fault-injection induced crash in the previous run, a file system check is forced. The bottom window shows another UMLinux machine just being installed with the installation routine of an out-of-the-box Debian Linux.

4 Configuring the Virtual Hardware

We group the hardware into the three main categories *computing core*, *peripheral hardware* and *external hardware*. All information about the virtual hardware of a single machine is gathered in a single directory.

UMLinux makes no changes to the way processes use the CPU, i.e. there is no simulated CPU, instead the real CPU is accessed directly. Thus it is not possible to configure the CPU. The memory management unit (MMU) is implemented using system calls which allow to map files into certain address ranges in main memory. It is always the same and cannot be configured. UMLinux simulates random access memory (RAM) with a memory-mapped file, the size of which is the size of the UMLinux machine's "physical RAM". It can be configured freely within the limits posed by the real hardware available on the host machine.

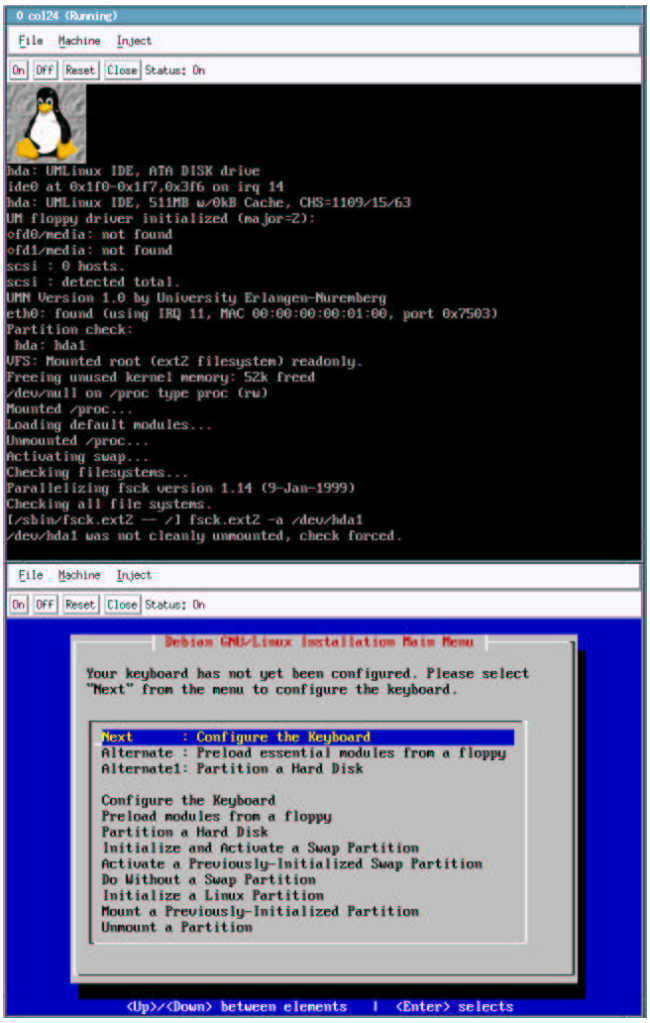

Fig. 2. Screenshot of the UMLinux user frontend

Peripheral devices mainly include storage devices, but also input/output devices and network interfaces. Block devices, such as harddisks, floppy and cdrom drives are implemented as files. The contents of these ”harddisk-files” can be created by installing an out-of-the-box Linux-distribution onto a file filled with only zeroes (the equivalent to an empty non-formatted harddisk of a certain size). It is also possible to create an image from a partition of a real harddisk and use it directly. Files implementing cdroms are simply image-files of real cdroms. It is also possible to access the cdrom of the host machine. The same holds true for floppy-drives. The number and size of the disks available to the virtual machine can be configured freely within the limits posed by the real hardware available on the host machine.

The console and keyboard device are not configurable.

The number of ethernet interfaces is configurable (the standard Linux kernel supports up to 16 interfaces). The number of free ports available on the host machine is usually large enough (there are up to 2^{16} ports on any machine running the IP protocol) to not pose a limit, even if several UMLinux machines are running on the same host. The networking hardware and routing setup is implemented by a separate process, called UM networking process (UMNP), which is configurable. When UMNP is running, the virtual machines are transparently integrated into the the local area network to which the host machine is attached.

5 Injecting Faults

The frontend (or Expect) needs to be configured to inject faults when a virtual machine is started. The description of the faults to be injected, the *fault load*, is configured per virtual machine using a simple textfile. For a large series of experiments, a script can generate a number of these files automatically. The graphical user frontend includes a simple fault description editor which supports interactive fault injection. For each fault to be injected at least the following information must be given:

Location: The tool currently supports the following fault locations:

- RAM
- CPU registers
- Blockdevices (harddisks, cdroms, floppy-drives, etc.)
  - failure of a number of consecutive bytes
  - device inaccessible
- Network interfaces
  - inability to send
  - inability to receive

Type: The fault type may currently be one of

- permanent
- duration
- transient

Activation Time: This is the time (in seconds, ≥ 0) at which the fault becomes active in the system. $t = 0$ is the time at which the virtual system boots.

Following the hardware categories defined in Sect. 4 we define fault types of the same categories, i.e. computing core faults, peripheral faults and external faults. For the first two categories, the following sections explain, what additional information is needed to fully describe those faults. External faults include power failures or failures of external networking hardware such as cables or switches. We are currently working on implementing these type of faults.

For technical information explaining *how* the faults are injected, please refer to [2].

5.1 Computing Core Faults

Computing core faults include RAM, MMU and CPU faults.

RAM faults include transient bit-flip and permanent stuck-at faults. In addition to the information listed at the beginning of this section, a RAM fault is defined by

Address: A valid address in the virtual machine's physical RAM (must fall on a word-boundary).

Bit: The number of the bit which flips (between 0 and 31, where 0 is the least and 31 is the most significant bit of the word).

CPU faults include transient bit flips or permanent stuck-at faults in registers. An effect may be instructions skipped or wrong branches taken. In addition to the information listed at the beginning of this section, a CPU fault is defined by

CPU-Number: The number of the CPU. This is always 0 for single-processor machines and ranges between 0 and the number of CPUs less one on multi-processor machines.

Register: A valid register of an Intel CPU (e.g. eip, eax, eds, esp).

Bit: The number of the bit which flips (between 0 and 31, where 0 is the least and 31 is the most significant bit of the register).

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.

5.2 Peripheral Faults

The following paragraphs treat peripheral faults.

In addition to the information listed at the beginning ot this section, a *block-device fault* is defined by

Device: A valid Linux block-device name, e.g. hda (IDE master on first controller, usually harddisk containing boot partition), hdc (IDE master on second controller, usually cdrom), fd0 (first floppy).

First Byte: (only for byte-wise failure) The number of the faulty byte.

Number of Bytes: (only for byte-wise failure) The number of faulty bytes following the first byte.

An active fault does not necessarily have to have an effect on the operating system at all, for example, when the blocks including the defect bytes on the harddisk are never accessed. If these blocks are accessed, the fault is noticed by the operating system, but unless the blocks contain data important to the operating system (such as filesystem information), there may be no further visible effects.

In addition to the information listed at the beginning of this section, a *network interface fault* is defined by

Device: A valid Linux network interface name such as eth0, eth1.

Injecting permanent faults into peripheral devices does not incur a significant overhead, since the virtual machine is stopped at entry and exit of every system call anyway, so that the tracer can redirect the system call to the UM kernel if necessary [2]. Additionally, only those system calls implementing UMLinux device 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.

6 Example Experiment

This sections describes a simple example experiment to show how to use 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 3 shows the percentage of different types of behavior observed in the example setup for the processor and harddisk faults. 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 Fig. 3. 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

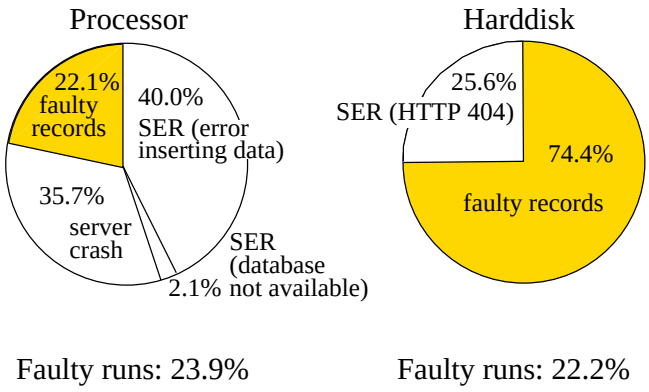

Fig. 3. Results of the experiments (faulty runs only)

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 Fig. 3. 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. 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.

7 Tool Demonstration

The graphical user frontend of UMLinux is better suited for presentation than the automatic experiment controller. We can show how to inject faults into one or more running UMLinux machines interactively using the graphical user frontend. The effect of some faults can be experienced directly by logging onto the UMLinux machines and checking

their behaviour with some commands. To view network faults, for example, set up two UMLinux machines each doing a `ping` to the other machine and then inject a network fault into the network interface of one machine. Flipping a bit in the program counter of the CPU will usually instantly panic the kernel. To view harddisk faults, running a lengthy search or copy over the complete harddisk while injecting the fault will trigger the output of a lot of error messages.

We can present UMLinux to a larger audience using a beamer and a prepared series of interactive fault injections using the frontend. A small audience could sit down in front of a few laptops running UMLinux and examine the frontend themselves or log onto a UMLinux machine to do a few tests.

Conclusion and Outlook

UMLinux is already a versatile tool for testing and fault injection and fully operational.

We are currently working on improving the simulation of the virtual hardware with the aim, of having to change as little as possible in the original Linux operating system to port it to our virtual hardware. We have already implemented VESA graphics hardware, IDE-Controller, real time clock, APIC, mouse and keyboard which can be accessed with the original Linux drivers. We are working on doing the same for the network interface.

We are currently implementing are medium scale experiment using the Linux Virtual Server [22] as a web frontend to a database to see how UMLinux scales for testing larger systems.

Acknowledgement

The research presented in this paper is supported by the European Community (DBench project, IST-2000-25425).

References

[1] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek. Fault injection experiments using FIAT. *IEEE Transactions on Computers*, 39(4):575–582, 1990.

[2] K. Buchacker and V. Sieh. Framework for testing the fault-tolerance of systems including OS and network aspects. In *Proceedings Sixth IEEE International High-Assurance Systems Engineering Symposium*, pages 95–105, 2001.

[3] J. Carreira, H. Madeira, and J. G. Silva. Xception: Software fault injection and monitoring in processor functional units. In *5th International Working Conference on Dependable Computing for Critical Applications*, pages 135–149, 1995.

[4] G. J. Carrette. Crashme. URL: http://people.delphi.com/gjc/crashme.html, 1996.

[5] Connectix Corporation. Virtual PC. URL: http://www.connectix.com/, 2001.

[6] 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. URL: http://www.laas.fr/DBench/, 2001.

[7] J. Dike. A user-mode port of the Linux kernel. In *5th Annual Linux Showcase & Conference, Oakland, California*, 2001.

[8] H.-J. Höxer, K. Buchacker, and V. Sieh. UMLinux - a tool for testing a linux system's fault tolerance. In *LinuxTag 2002, Karlsruhe, Germany, June 6-9, 2002*, 2002.

[9] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into VHDL models: The MEFISTO tool. In *Proceedings of the 24th IEEE International Symposium on Fault Tolerant Computing*, pages 66–75, 1994.

[10] G. Kanawati, N. Kanawati, and J. Abraham. FERRARI: A tool for the validation of system dependability properties. In *Proceedings of the 22th IEEE International Symposium on Fault Tolerant Computing*, pages 336–344, 1992.

[11] N. Kropp, P. J. Koopman, and D. P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proceedings of the 28th IEEE International Symposium on Fault Tolerant Computing*, pages 230–239, 1998.

[12] K. Lawton. Plex86. URL: http://www.plex86.org/, 2001.

[13] 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. Computer Science Technical Report 1268, University of Wisconsin-Madison, 1995.

[14] M. Rodríguez, F. Salles, J. C. Fabre, and J. Arlat. MAFALDA: Microkernel assessment by fault injection and design aid. In *3rd European Dependable Computing Conference*, pages 208–217, 1993.

[15] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The simos approach. *IEEE Parallel and Distributed Technology*, Fall, 1995.

[16] V. Sieh. Fault-injector using UNIX ptrace interface. Internal Report 11/93, IMMD3, Universität Erlangen-Nürnberg, 1993.

[17] V. Sieh and K. Buchacker. Testing the fault-tolerance of networked systems. In U. Brinkschulte, K.-E. Grosspietsch, C. Hochberger, and E. W. Mayr, editors, *International Conference on Architecture of Computing Systems ARCS 2002, Workshop Proceedings*, pages 37–46, 2002.

[18] V. Sieh, O. Tschäche, and F. Balbach. VERIFY: Evaluation of reliability using VHDL-models with integrated fault descriptions. In *Proceedings of the 27th IEEE International Symposium on Fault Tolerant Computing*, pages 32–36, 1997.

[19] Source Forge. Bochs IA-32 Emulator Project. URL: http://bochs.sourceforge.org/, 2001.

[20] Virtutech Inc. simics. URL: http://www.simics.com/, 2001.

[21] VMware Inc. VMware. URL: http://www.vmware.com/, 2001.

[22] W. Zhang. Linux virtual server for scalable network services. In *Ottawa Linux Symposium 2000*, 2000.