A Software Layer for IDE Disk Fault Injection

To appear in *Systems Lacking Originality Workshop 2005*
Jake Adriaens, Dan Gibson
University of Wisconsin-Madison Department of Electrical and Computer Engineering
{jtadriaens, degibson}@wisc.edu

Increasing demands for reliability, fault isolation, and fault tolerance have emphasized the need to study systems under failure conditions. Commodity disk systems have been studied for many years with simple assumptions about how they fail. To provide enabling technology for new research in more reliable file systems, disk systems, and I/O interfaces, we present a flexible, software-based fault injection system, targeted for commodity disks running in conjunction with a commodity operating system. We present five sophisticated but general disk failure models, and a simple interface to allow arbitrary faults to be injected rapidly and without permanent hardware damage. We also evaluate the performance of a common file system utility to detect these faults, and present a summary of our findings.

1 Introduction

As commodity hardware and software becomes more commonplace in high-end systems, greater attention must be paid to how devices fail and how systems react to and tolerate device failure. Applying this concept to disks is not a new idea—redundant arrays of inexpensive disks (RAID) have been using commodity hardware with its failure in mind for many years [14].

Disk failure has often been a woeful observation of file system, RAID, and vulnerability studies [1,3,6,7,11,12], but models of disk failure in research have been limited. To further research into fault-tolerance and fault-recovery measures and how they might be applied to commodity hardware, an improved model of disk failure and a method to simulate failures is needed.

To this end, we have developed a software implementation of disk fault injection, targeted to the x86/Linux community. Software-implemented fault injection (SWIFI) has been successfully used in many other fields, including networking [13,20], processors [25], and even kernels [8,23]. We continue that tradition with an implementation of fault injection for a commodity system that could be used for real workloads: the EIDE I/O system on a Linux x86-based machine.

Our approach uses a software layer that sits logically between the Linux IDE disk driver and all higher-level software. In practice, this layer is actually a demand-loaded module within the kernel's code space, that interposes on disk read and write events. Faults are defined at a high level in human-readable format, parsed by a specially designed parsing process, and passed into the kernel for manifestation. Faults are then excited through normal disk read and write operations. The operation of the injector is transparent to the driver's implementation below the module, and equally transparent to the file system above, except when injecting faults into the system.

In general, core drivers—like the Linux IDE driver—are developed with simplicity, reliability, and fault tolerance in mind. It is not a simple process to inject meaningful faults into these systems in a manner that does not destabilize the system built on top of them, and without adversely affecting performance. Worse still, fault models tend to be catastrophic in nature, or only partially developed. Disk manufacturers are reluctant to provide accurate models of failure for business reasons, so those in the academic community must approximate these failures with models as general as possible, while still relating their systems to real hardware.

We present herein our implementation of SWIFI for commodity disks. While we focused on x86-based Linux machines with IDE-based disks, the same techniques, if not the same implementation, could easily be applied to other architectures, operating systems, and disk interfaces.

The remainder of this paper is organized as follows: In Section 2 we review work related to

SWIFI for disks; Section 3 discusses our fault models; Section 4 describes our implementation in detail. Section 5 presents our evaluation methodology; Section 6 presents our qualitative and quantitative results. Section 7 concludes this paper.

2 Related Work

Software fault injection techniques have been used extensively in computer science to evaluate fault-tolerance of systems and accompanying fault-detection, fault-diagnosis, and fault-recovery mechanisms. While our work applies transparent software fault injection to disks, the technique has been applied to nearly all other realms of computer systems. Jarboui, Arlat, Crouzet and Kanoun employ three fault injection techniques to the kernel of the Linux operating system [9]. Each of these techniques corrupts parameters to various calls within the kernel, thereby simulating errors and misbehaviors that may occur within higher-level software. These techniques have very low overhead, and are entirely invisible when properly implemented. Our approach to fault injection uses a nearly identical method, but applied to the Linux IDE driver alone. Gu et. al. [8] conducted a similar study of Linux kernel behavior by injecting faulty branch instructions into the kernel's code. The work done by Gu et. al. provided excellent foresight on what behaviors to expect when injecting faults into low-level software.

Stott, Ries, Hsueh and Iyer use a SWIFI-based injection mechanism to corrupt instructions passed to a network processor [20]. Through this corruption, the authors can infer how some network services tolerate faults in network communication. These faults included unexpected packet drops, corruptions of packets, and unintended broadcasts or retransmissions. A similar network-oriented fault injection scheme was used by Nagaraja et. al. [13].

Kaâniche, Romano, Kalbarczyk, Iyer and Karich use a disk fault injection method to evaluate fault tolerance and performance in faulty conditions in a commercial RAID system under simulation [10]. This type of study makes heavy use of disk fault injection techniques, and examples of disk fault injection used in this context abound. However, we note that most of these injection schemes only produce simulated *detected* (or easily-detected) faults. In practice, we believe that *undetected* (and hard-to-detect) faults are of equal significance.

The FINE system described by Kao et. al. employs software techniques to introduce software faults and simulated hardware faults into the UNIX kernel [23]. Among these faults are disk faults, modeled as bus failures. The focus of [23] is the tracking of *fault propagation*, after injection and manifestation. Specifically, the authors explore the effect of faults from a desire to improve fault isolation measures.

Prabhakaran et. al. [15] are currently working on improved disk fault injection models and how they might be used to evaluate file system fault tolerance and detection. Additionally, a greater understanding of file system behavior in the presence of faults grants insight on where improvements may be possible. Much of our work on type-sensitive failures closely parallels this work. Our choice of injection point is also very similar—[15] implements a *pseudo-driver* between the file system and lower-level code, while we make modifications to the IDE driver itself. In practice, our low-level approach enables us to model faults in a nearly identical manner as the use of a pseudo-driver would allow. Since much of our implementation is encapsulated in a Linux module, one might think of our injection module as *pseudo*-pseudo-driver.

3 Fault Models

In our exploration of previous work in disk fault injection, we found that a common trend was to use simple failure models [10,14]. Simple models are easily implemented and validated, and allow other studies to continue without the overhead of developing more general injection systems. As the primary focus of our work was to produce such a system, we devoted much of our time to developing failure models that reflect less catastrophic events than those of the `failstop` model, but still had general application.

We also noted that disk manufacturers are not forthcoming of the failure characteristics of their products, for obvious business reasons. It is still possible to find information on specific disk failures ([6], for instance), though many of these descriptions are non-technical, or are too specific

(too closely coupled to a certain hardware configuration). We have drawn on existing models [15], personal experiences with disk failures, and intuition to develop general disk failure models, which are intended to model hardware failures in the physical media, controller malfunctions, motherboard design flaws, and driver bugs, and other errors.

3.1 Specific Fault Models

Our implementation supports five specific fault types, in addition to the traditional model of disk failure, "`failstop`" in our context. We classify our other five fault types as "fail-wrong," as they simulate erroneous behavior without flagging error conditions within the IDE driver; hence the system *fails* in the *wrong* way.

Unless otherwise noted, each of the faults described below is applicable to a single sector or a sector range (of course, multiple faults can be used to simulate more sporadic fault patterns). Additionally, each fault has a *manifestation probability* associated with it, which is consulted to randomly determine when the fault may manifest itself. Of course, faults may have a probability of 1.0, which corresponds to "always present." Accompanying this probability, there is a probability mutator value that is bitwise XORed with the probability after manifestation. This allows probabilities of manifestation to change, and is used by our transient fault models extensively.

- `sectorfail` faults model the general failure of a specific sector or sector range on the disk to reliably store data. Data read from blocks under a `sectorfail` fault will not return correct data. Specifically, they will return data as specified by the fault injection module—options include constant values (e.g. the block reads as all-zero), permutations of correct values (such as byte shifting as described in [15]), logical permutations, such as AND or XOR with a constant value, or simply randomization. Our primary inspiration behind this fault type is the effect of head-to-platter impacts in drives due to jostling or accidental vibration.

- `sectorwrong` faults model problems in on-disk block addressing, which can arise from either driver malfunctions or hardware failures. Reads and writes to a sector affected by `sectorwrong` are redirected to another block. The nature of this redirection varies, depending on the fault specification provided in the fault list. This redirection could be to a specific location, an offset of the intended destination, a bitwise shift, randomized, or otherwise mutated. We intend this error to model potential disk controller problems, head alignment problems, or even vulnerability of memories to random corruption.

- `sectorro` is unlike the previous two failure models in that it does not modify a request that has or will take place—it simply causes all writes to affected sectors to be ignored by the IDE driver. This has the effect of producing read-only sectors on the disk. Another name for this failure model is the *phantom write* [15]. It is inspired by damage to physical storage media, which can cause certain regions of disk to become read-only.

- `transaddr`, or "transient address fault," is very similar to the `sectorwrong` failure model described above. It permutes addressing information using an identical technique, but also employs a probability mutator to cause the fault to disappear after the first manifestation. Through personal experience, we have observed this kind of faulty behavior due to head positioning errors and degrading performance of arm servos, or even physical wear of the drive. Some older drives were belt-driven, and this belt could wear over time. We considered using the system speaker to emulate the sound of this failure, but decided that it held little scientific merit.

- `transdata` is the transient version of the `sectorfail` fault, or the data-affecting version of the `transaddr` fault. It permutes data in a manner identical to that of the `sectorfail` fault, but disappears after the first manifestation of the fault. We intend this fault to model disk controller

malfunction, head synchronization problems, or on-disk caching inaccuracies.

- `failstop` models the complete, catastrophic failure of the disk. This fault was the simplest to implement, and the easiest to detect. It causes the disk to ignore all requests, which models a great number of hardware problems.

We had intended to model a sixth fault type, `stuckat`, which would have represented the failure of a single line on the EIDE bus (inspired from VLSI fault models [4]). However, it quickly became evident to us that `stuckat` would behave almost identically to `failstop`, so we abandoned this model.

4 Implementation

Our software fault injection system is comprised of three core components: a fault control application, in-kernel calls and special-purpose code, and an injection module. In brief, the control application is a user-level process written using standard libraries, which takes as input a list of faults to be inserted. This list is parsed and then passed to the injection module, which resides in kernel space. In-kernel and in-driver "upcall" functions allow the module to interpose on every IDE disk command, to perform fault injection if appropriate.

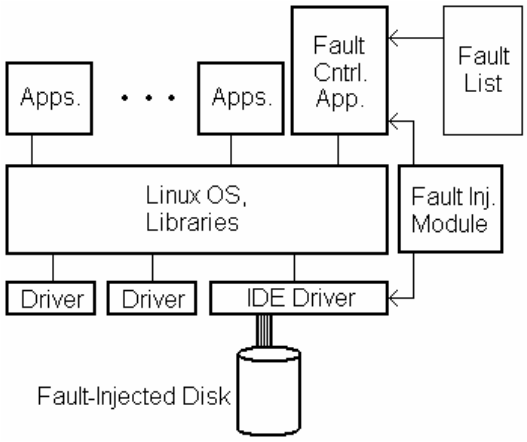

Figure 1 - Interaction of Fault Control Application, Injection Module, and Driver

Before describing each of these systems in detail, we briefly present our observations of the Linux IDE driver that enable our implementation to function as it does.

4.1 Linux IDE Driver Observations

At first glance, the in-kernel IDE driver code for our version of the Linux operating system (Kernel version 2.4.29, from the Slackware [19] distribution) seems very complex. In fact, there are many thousands of lines of code in the source tree of the IDE driver. Our first observation is that the hard disk-related driver code is contained almost entirely in `ide.c` and `ide-disk.c`, the main exception in the form of some calls to functions in `ide-io.c`.

Working only with these two files greatly simplified our implementation, but together they still account for thousands of lines of C code. We explored the functions contained in these files through insertion of `printk` statements (analogous to the user-level `printf` statement), and observations of logged outputs. Eventually, we noticed that all IDE read and write requests have a common "choke-point" in their execution: the `ide_do_rw_disk()` function. While similar functions exist throughout the driver, our observations indicate that only this function was in use in our system.

We also made an observation of the multitude of structures that are visible within the above-mentioned function. Specifically, we noted that the `request` structure contains all useful information about an IDE command, including its type (e.g. "read" or "write"), location on disk, and pointer to data in the case of write commands, or a pointer to the intended destination in memory in the case of read commands.

Often, the `request` structure is accompanied by an `ide_drive` structure. This structure contains useful information about the drive associated with the request, including the drive's name, interfaces, addressing information, etc. We use both the `request` and `ide_drive` structures to determine information about disk activity, and may make modifications to the `request` structure as part of the injection process.

4.2 Fault Control Application

The fault control application resides in user space. Its purpose it to parse human-readable fault descriptions into a generalized fault definition structure. Any fault in our fault model may be specified in the fault control application, and faults may be injected on a per-drive basis, simply by specifying the drive's device name under the Linux operating system ("`hdc`," for instance). This application was critically necessary for our project, as the obvious alternative to using a control application is to either hard-code the faults into the kernel (which requires a long compile cycle and reboot to change the fault configuration, and may cause permanent instability), or to include static definitions in the injection module (which, again, would require a recompile and explicit re-loading of the module to change the fault configuration).

Additionally, the control application enables faults to be extracted as well as injected. This was a very useful feature during development, as early implementations caused unusual and often destabilizing side effects, due to bugs in kernel-space code. Note that this is separate from the mechanism by which transient faults are modeled.

4.3 In-kernel code

We have made surprisingly few changes to the low-level kernel code. Within the IDE drivers, we insert calls to the fault injection module in two locations. The first of which occurs before a command is passed to the drive, in the `ide_do_rw_disk()` driver function. The purpose of this call is to provide the opportunity to modify data to be written, in the case of a write command, or to modify the disk addressing information (in the `request` structure) prior to either read or write commands.

The second call to the module occurs only for read events, within the event handler that is called on the completion of a disk access. This upcall exists to allow modification of data that has been read from the disk.

In addition to these two upcall sites, we have added a small amount of special-purpose code to the IDE driver. Specifically, this code implements our `failstop` model, and prematurely ends disk write commands in the case of the `sectorro` fault. Other modifications to kernel space were made only to add system calls, which result in small modifications to a number of files in the kernel source tree.

4.4 Fault Injection Module

The role of the fault injection module is two-fold:

- Accept faults from the user-level control process via added system call `insert_faults()`
- Apply faults as defined in the current fault list when modifying IDE commands and structures

The simpler of these two operations is the former. To accept new faults, the module simply adds new `fault_entry` structures (defined by our implementation) to the end of the current fault list, in the case of insertion. The same system call is used to remove faults, which is accomplished simply by dereferencing the pointers associated with the faults.

Fault application may occur during upcalls from the IDE driver. The module iterates through the list of injected faults, checking the affected sector range as defined in the `fault_entry` structure against the sector(s) involved in the IDE request (from the `request` structure). If a faulted range matches the currently referenced sector(s), the accompanying probability of fault manifestation is consulted. At this point, the fault is probabilistically applied to the `request` structure (that is, the fault may not be applied, based on random chance if the fault is not "always" present). If the fault is excited, the probability of manifestation is `XOR`-ed with the probability mutator of the fault. This is the mechanism by which transient faults "disappear" in our injection system: we simply set the probability mutator equal to the initial probability of manifestation—after the first sector range match, the probability of fault manifestation becomes zero.

Application of faults is straightforward—all fields modified belong to the `request` structure. Depending on the fault configuration, some mutations of the `request` structure will occur.

5 Methodology

Our primary means of evaluating our fault injection system was to inject single faults (at known, predictable locations) and observe the faulty behavior induced by them. This is a straightforward process, and does not have many of the same challenges as those in fault detection [11,12]. However, many of the same techniques are used, mostly to determine that fault injection does not have any readily observable effects on regions of disk that should remain unaffected.

We evaluated our machine on an Intel Celeron ® -based machine, which has a processor clock frequency of 1.4 GHz. Our main memory size was 64 MB. The size of our primary disk was 15 GB, with a swap space of 4 GB. We employed a second disk for fault injection; the size of the disk used for fault injection was 4 GB. We used Linux 2.4.29 from the Slackware [19] distribution as our operating system.

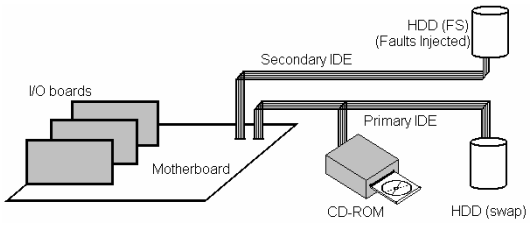

Figure 2 - System Configuration

We also employed built-in hardware cycle counters to determine the added access latency of our software. These tests were run using a user-level microbenchmark, which employed both file-system based I/O benchmarks and raw disk access functions. The validation of our cycle counters was performed in an earlier, unpublished study.

In addition to verifying the intended effect of our various fault models, we conducted a small experiment with the EXT-2 file system utility `mke2fs` [22]. Specifically, we attempted to build a file system on top of a faulty disk, with many different faults inserted. Our findings for this small study are presented below, in the next section.

The `sectorfail` and `transdata` faults were evaluated using similar methods, as they have similar effects. However, since we must ensure both that `transdata` disappears after manifestation and `sectorfail` does not, we were able to use the same microbenchmark for both faults. Specifically, these faults can be observed by the following steps:

1. Write data to a known location (a file), without injecting any faults. The data of this file is known a priori, so reading from this file becomes a simple way to detect data corruption.
2. Flush the file cache. This is accomplished in Linux by `umount`ing and `mount`ing our test disk.
3. Inject a `sectorfail` or `transdata` fault onto our test disk at the sector corresponding to the file from 1, using the control application.
4. Read from the file from 1; at this point, the observed data should be incorrect. The file length, however, is correct, as it is stored separately from the file under EXT-2. So, using the utility `cat` will show the presence of this fault.
5. Flush the file cache. This ensures that faulty data is not cached, so a second read can occur.
6. Read again from the file from 1. For a `sectorfail` fault, the data will again be corrupted. `transdata` faults will not further corrupt the data, and the correct value will be returned.

Verification of the `sectorwrong` and `transaddr` faults was accomplished by a separate approach. As each of these modify the location information associated with a request, both the manifestation and the correct re-positioning of the request must be observed. We verify these faults by:

1. Write data to two known locations (two files).
2. Flush the file cache.
3. Inject a `sectorwrong` or `transaddr` fault to the location of one of the files' data. This fault should reassign the sector corresponding to one file to the other, so reads from the affected file will effectively read the other file. This ensures that not only is the addressing information corrupted, but it is corrupted in the expected manner.
4. Read from the affected file. In both the `sectorwrong` and `transaddr`

cases, the output of `cat` should match the "unaffected" file.

5. Flush the file cache. Again, we must re-read the block to ensure that `transaddr` has disappeared and `sectorwrong` has not.
6. Read again from the affected file. In the `transaddr` case, we now read the correct data. In the case of `sectorwrong`, the value from the unaffected file is read a second time.

The following steps verified the `sectorro` fault:

1. Write known data to a file.
2. Flush the file cache.
3. Inject a `sectorro` fault onto the disk, on the sector corresponding to the file from 1.
4. Write random data to the file from 1.
5. Flush the file cache.
6. Read from the file from 1, observe the same data written in step 1—the write from step 4 should not have had an effect on the state of the disk.

Evaluation of the `failstop` fault was extraordinarily simple—we needed only observe that no reads or writes occur when `failstop` is injected. From the perspective of user-level processes, this fault often manifests itself as a "kernel panic," in which the system comes to an unexpected halt.

6 Results

Our measurements of added latency due to injection indicate that there is not a significant or consistently measurable performance impact associated with disk SWIFI. A summary of the average runtimes of many (100) repeated reads to the same block under our injection system is detailed in Table 1.

	Mean Access Time (ms)	**Std. Dev.**
No injection present	3.025	0.075
Unaffected Sector	3.020	0.076
Affected Sector	3.044	0.081

Table 1 – Runtime overhead of disk fault injection

The first row of Table 1 shows read performance without any kernel modification. The second row depicts performance with all of our modifications in place, but for IDE traffic that does not excite faults. The third row shows average operational latency for read commands that excite faults. Our fault injector does not incur a significant performance penalty—even in the case of an affected sector, the measured increase in access time is much less than the standard deviation over 100 accesses.

We also evaluated the effect of the `mke2fs` command from the EXT-2 program suite [22] under our fault injection system. We were surprised by our results, which is summarized in Table 2.

`mke2fs` params.	**Faults (number)**	**Detected?**
`-v`	`sectorfail` (10)	No
`-v -c`	`sectorfail` (10)	No
`-v -c -c`	`sectorfail` (10)	Yes
`-v`	`sectorfail` (10000)	No
`-v -c`	`sectorfail` (10000)	No
`-v -c -c`	`sectorfail` (10000)	Yes
`-v`	`sectorwrong` (10)	No
`-v -c`	`sectorwrong` (10)	No
`-v -c -c`	`sectorwrong` (10)	No
`-v`	`sectorwrong` (10000)	No
`-v -c`	`sectorwrong` (10000)	No
`-v -c -c`	`sectorwrong` (10000)	No
`-v`	`sectorfail` (3, superblocks)	No
`-v -c`	`sectorfail` (3, superblocks)	No
`-v -c -c`	`sectorfail` (3, superblocks)	Yes

Table 2 – `mke2fs` effectiveness in the presence of faults

In Table 2, three parameter sets are passed to `mke2fs`. Specifying the `-c` parameter indicates that a quick, read-only check should be used to

check for "bad blocks." Specifying the `-c` parameter twice indicates that a more sophisticated, alternating-pattern write/read test should be used. The `-v` option is used only to produce verbose output.

In general, the file system generation utility is very vulnerable to disk faults. While it is perhaps not surprising that `mke2fs` should succeed with a small number of faults (10), most would agree that in the presence of a large number of consecutive faults (10000), `mke2fs` should at least generate a warning. However, in the cases above marked as not detected, `mke2fs` succeeds with no apparent warning.

We note that `mke2fs` succeeds in the presence of all `sectorwrong` faults, and only during the most careful checks does it fail to succeed when `sectorfail` faults are inserted. This includes locating faults onto the superblocks of the file system. Due to its test method, we expect `mke2fs` to succeed with an arbitrary number of `sectorwrong` faults, as described below.

Without specifying any parameters (except `-v`), `mke2fs` simply attempts to write the necessary blocks to create the file system. However, in the presence of both `sectorfail` and `sectorwrong` faults, writes to any block will succeed. Hence, the consistent failure to detect faults with this parameter configuration is expected.

The `-c` parameter causes `mke2fs` to use a simple test of all blocks—not only those needed to create the file system. The program reads each file from the disk once, in sequential order, without examining the data (of course, `mke2fs` cannot know against what data it should compare!). Since neither `sectorfail` nor `sectorwrong` necessarily generate errors on manifestation, `mke2fs`'s simple test is inadequate to detect these errors, as well.

The `mke2fs` utility uses an alternating-pattern write/read test to verify each block (one at a time) on a disk for "bad blocks" as its most careful test (denoted by `-v -c -c` in the table). Specifically, `mke2fs` writes the hex pattern 0xAA to each byte of a block, then reads that pattern from the disk. Once all blocks have been tested in this manner, the value 0x55 is written and read. This effectively tests the disk for single-bit stuck-at faults [4], but does not test for addressing faults. In effect, the test may be run on the same sector more than once when under the effect of a `sectorwrong` fault, and the test succeeds simply because the block that has been accessed can be written and read without error. No test exists to ensure that the same block is not accessed more than once per iteration. Techniques for testing this fault type have been developed for memories [4], but are far too slow to be implemented for disks, making `sectorwrong` a very dangerous (and powerful) fault indeed.

7 Summary

We have implemented a software IDE-based disk fault injection system for the Linux operating system on x86-based computers. We present six fault models, which can be injected via a simple console interface and represent a wide range of potential hardware and software failures. Each of these failure models is inspired by real hardware or software failure, and has the desired effect on IDE read and write requests.

We also evaluated the runtime overhead of our software fault injector, and found the overhead to be so small as to be immeasurable with any accuracy. Specifically, we found no significant runtime overhead of our injector as compared to the injection-free (correct) case.

We evaluated the ability of the `mke2fs` file system utility to detect latent disk faults we injected with our system. We show that careful checks performed by `mke2fs` can detect failed disk sectors, but cannot detect sector addressing faults that may exist in low-level software or hardware.

Acknowledgements

We thank our advisor and reviewer of this work, Remzi Arpaci-Dusseau, for his tireless efforts to answer our seemingly endless questions during this project.

We also extend our thanks to the developers of the Linux kernel `panic()` functionality, for destroying our primary disk on our test hardware during an unexpected panic. Without you, we wouldn't have our wonderful 25 GB paperweight.

References

[1] J. Arlat, Y. Crouzet and J.-C. Laprie, "Fault Injection for Dependability Validation of Fault-Tolerant Computing Systems," In: *Proceedings of the 19th International Symposium on Fault-Tolerant Computing*, June 1989.

[2] Remzi Arpaci-Dusseau, Personal communication.

[3] J. H. Barton, E. W. Czeck, Z. Z. Segall and D. P. Siewiorek, "Fault Injection Experiments using FIAT," In *IEEE Transactions on Computers*, Volume 39, Issue 4, April 1990.

[4] Michael L. Bushnell and Vishwani D. Agrawal. *Essentials of Electronic Testing for Digital, Memory & Mixed-Signal VLSI Circuits*. Kluwer Academic Publishers, Massachusetts. 2000.

[5] João Carreira, João Gabriel Silva. "Why do some (weird) people inject faults?" In: *ACM SIGSOFT Software Engineering Notes*, Volume 23, Issue 1, 1998.

[6] Data Clinic, The. "Hard Disk Failure," http://www.dataclinic.co.uk/hard-disk-failures.htm, 2004.

[7] R. Green. "EIDE Controller Flaws," http://mindprod.com/eideflaw.html, 2005.

[8] Weining Gu, Z. Kalbarczyk, Ravishankar Iyer, and Zhenyu Yang. "Characterization of Linux Kernel Behavior Under Errors," In: *Proceedings of the International Conference on Dependable Systems and Networks*, June 2003.

[9] Tahar Jarboui, Jean Arlat, Yves Crouzet and Karama Kanoun, "Experimental Analysis of the Errors Induced into Linux by Three Fault Injection Techniques," In: *Proceedings of the International Conference on Dependable Systems and Networks*. 2002.

[10] M. Kaâniche, L. Romano, Z. Kalbarczyk, R. Iyer and R. Karich. "A Hierarchical Approach for Dependability Analysis of a Commercial Cache-Based RAID Storage Architecture," In: *Proceedings of the 28th International Symposium on Fault-Tolerant Computing*, June 1998.

[11] Hannu H. Kari, Heikki Saikkonen and Fabrizio Lombardi. "Detecting Latent Sector Faults in Modern SCSI Disks," In: *Proceedings of the 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, January-February, 1994.

[12] Hannu H. Kari, Heikki Saikkonen and Fabrizio Lombardi. "Detection of Defective Media in Disks," In: *Proceedings of the IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, October 1993.

[13] Kiran Nagaraja, Xiaoyan Li, Ricardo Bianchini, Richard P. Martin, Thu D. Nguyen, "Using Fault Injection and Modeling to Evaluate the Performability of Cluster-Based Services," In: *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

[14] David A. Patterson, Garth Gibson, and Randy H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," In: *Proc. ACM SIGMOD Conference*, June 1988

[15] Vijayan Prabhakaran, Nitin Agrawal, Lakshmi Bairavasundaram, Haryadi Gunawi, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, "Rusty File Systems," Unpublished document. 2005.

[16] H. Reiser. "ReiserFS," www.namesys.com, 2004.

[17] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers, 2nd Edition*, O'Reilly. June, 2001.

[18] A. Silberschatz, P. Galvin, G. Gagne. *Operating System Concepts, 6th Edition.* John Wiley & Sons, Inc. New York, 2002.

[19] Slackware. www.slackware.com

[20] David T. Stott, Greg Ries, Mei-Chen Hsueh and Ravishankar K. Iyer. "Dependability Analysis of a High-Speed Network Using Software-Implemented Fault Injection and Simulated Fault Injection," In: *IEEE Transactions of Computers*, Vol. 47, Issue 1. January, 1998.

[21] T. K. Tsai and R. K. Iyer. "Measuring Fault Tolerance with the FTAPE Fault Injection Tool," In: *The International Conference on Modeling Techniques and Tools for Computer Performance and Evaluation*, September 1995.

[22] T. Ts'o. "Ext2fs Home Page," http://e2fsprogs.sourceforge.net./ext2.html, 2005.

[23] W. I. Wao, R. 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*, Volume 19, Issue 11, November 1993.

[24] C. R. Yount and D. P. Siewiorek. "A Methodology for the Rapid Injection of Transient Hardware Errors," In: *IEEE Transactions on Computers*, Volume 45, Issue 8, August 1996.

[25] C. R. Yount and D. P. Siewiorek. "Sofware-Implemented Fault Injection of Transient Hardware Errors," In: *Foundations of Dependable Computing: Models and Frameworks for Dependable Systems*, G. M. Koob, C. G. Law, eds., chap. 3.1. Kluwer Academic Publishers.

Signatures

Jake Adriaens

Dan Gibson
