FIG: A Prototype Tool for Online Verification of Recovery Mechanisms [*]

Pete Broadwell
University of California, Berkeley
Soda Hall
Berkeley, CA 94720
pbwell@cs.berkeley.edu

Naveen Sastry
University of California, Berkeley
Soda Hall
Berkeley, CA 94720
nks@cs.berkeley.edu

Jonathan Traupman
University of California, Berkeley
Soda Hall
Berkeley, CA 94720
jont@cs.berkeley.edu

ABSTRACT

Network applications of the future will require advanced mechanisms for automatic failure recovery in order to provide an acceptable quality of service. Because of this requirement, there is a need for tools that can inject simulated faults to verify a system's fault detection and recovery methods. We present a set of objectives that such a fault injection tool must meet, and describe a prototype we have developed that meets these criteria. This tool, `FIG`*, is capable of selectively introducing and logging errors at the application/library boundary on a running system.*

Our expectation is that `FIG` *will be used for software development and verification, as well as for on-site testing of production systems. We use a working version of this tool to test the behavior of several common UNIX applications under simulated failures and offer suggestions from these tests on how to develop software with a higher degree of failure recoverability.*

Keywords

Fault injection, recovery-oriented computing, online testing, `glibc`, extensibility

1. INTRODUCTION

1.1 Motivation

The highly complex and dynamic nature of the next generation of network applications will make developing and administering these programs by traditional means impracticable. Efforts are currently underway to build adaptive, self-managing capabilities into next-generation business network and Internet applications [4], [7].

A vital component of such software adaptability is the ability of a system to detect and recover quickly from irregularities in its operating environment. Recovery-Oriented Computing [1], an ongoing project at the University of California, Berkeley, is dedicated to researching new methods for providing this type of recoverability in contemporary Internet-based applications. One fundamental component of a recovery-oriented design framework is the use of tools that can introduce a variety of simulated faults into a running system in a controlled manner. These tools can then be used to evaluate the effectiveness and integrity of recovery methods.

Tools for simulating faults have long been used in the development of fault-tolerant systems. In addition, they often appear in the later phases of testing for commodity software packages. However, fault injection is notably absent in the development process of many Internet applications, due to the abbreviated development cycle these projects tend to adopt. As a result, the recovery methods in these programs, if they exist at all, often go to market untested.

By advocating the development of enhanced failure recovery techniques for Internet applications, recovery-oriented computing makes the use of fault injection an integral part of the software development cycle. Moreover, this philosophy calls for the simulation of failure modes, such as user error and excessive system load, that are not normally associated with traditional fault-tolerant program development.

Possibly the most controversial aspect of fault injection for recovery-oriented systems is the idea that there may be benefits associated with introducing simulated faults at a constant, low frequency into running, online production systems. Essentially, a system furnished with a suite of efficient failure-recovery tools may actually see an increase in overall availability if its fault-injection subsystem successfully exposes latent software errors that would not otherwise have been detected. This conjecture is based on research done on complex systems in a variety of engineering fields, which indicates that many large-scale failures are actually the result of numerous small "latent" faults that go undetected during early design and testing phases, but later combine to cause a catastrophic failure [11].

Further discussion on the subject of latent errors is be-

[*]This work was supported in part by the Defense Advanced Research Project Agency of the Department of Defense, contract no. DABT63-96-C-0056, the National Science Foundation, grant no. CCR-0085899, NSF infrastructure grant no. EIA-9802069, Alocity, Microsoft, and the California State MICRO Program. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

yond the scope of this paper. However, consideration of the issue yields the directive that fault injection tools for recovery-oriented systems should be capable of running in an unobtrusive manner on a production system.

1.2 FIG: Fault Injection in glibc

Given the criteria listed above, our goal was to develop a lightweight, extensible prototype tool for testing the recoverability of software packages against a variety of external failures. The result of our efforts is `FIG`, a tool that is capable of injecting and logging errors at the application/library boundary with minimal configuration and run-time overhead.

Currently, `FIG` runs on UNIX-like operating systems and operates by interposing a library between the application and other function libraries that intercepts calls from the application to the system. When a call has been intercepted, `FIG` then chooses, based on testing directives from a control file, whether to allow the call to complete normally or to return an error that simulates a failure of the operating environment.

We believe that `FIG` is a suitable prototype of the class of fault injection tools called for in the recovery-oriented computing philosophy. As described in Section 2, `FIG` is simple to install, easy to enable or disable, and incurs a minimal amount of overhead while running. It can be directed to test a single application or a group of applications, and can either introduce faults into a small, targeted group of system functions to simulate the loss of a particular component, or else inject a wide variety of faults to test the behavior of the system under large-scale failures.

We hypothesize that the use of `FIG` as a development tool will contribute to the genesis of recovery-oriented applications. If software authors use `FIG` as a tool for testing their applications against failures in the system environment, the result of their efforts will be programs that are more robust and resilient. In addition, the use of `FIG` on a running production system may expose software errors that were not detected during testing, due to the differing natures of the testing and production environments. This latter application of `FIG` requires that the system must already have in place recovery mechanisms like automatic failover in order to mask the errors that `FIG` exposes.

1.3 Related Work

A wide variety of research work on fault injection already exists in the software engineering and fault-tolerant computing fields, and it is important to examine these past projects in order to establish where `FIG` stands in relation to them. Three general categories of existing fault injection tools are hardware-implemented fault injection, simulator-based fault injection and software-implemented fault injection, known as SWIFI. `FIG` is of the latter type.

Perusal of the "related work" sections of most papers on SWIFI techniques for fault tolerance yields a virtual alphabet soup of project names and acronyms, such as FIAT, FERRARI, FINE, FTAPE, DOCTOR, MAFALDA, HYBRID, ORCHESTRA and GOOFI [2]. Most of these projects were written to perform fault injection on a single hardware/software platform. In some cases, this platform is a real-time system that runs provably error-free software and operates in an environment requiring minimal human interaction and maintenance. As a result, the majority of the injected fault types are very low-level hardware faults, such as processor or memory failures that may corrupt a process's data or run-time image.

By comparison, our fault injection tool is designed to run on widely-used commodity platforms such as those supported by UNIX, and the types of failures we seek to inject may better be described as "perturbations" of the operating environment. These include problems caused by hardware failures, misbehaving user processes, human operators and system load spikes.

The body of research regarding these types of errors is much more limited. The majority of such projects, like the Ballista [8] and Fuzz [10] tools, seek to inject errors into a software module only at the user interface level. This interface level is perhaps the most obvious choice for targeted testing, due to its high visibility and accessibility. However, it does not include all of the interactions that take place between an application and the surrounding software environment.

Our model of fault injection is most similar to the expanded model endorsed by Whittaker [13] and later implemented in Holodeck [5], which calls for integrated testing of interfaces other than the application's external user interface. Examples of such interfaces include interactions between an application and the operating system, or between an application and other function libraries. A tool that can inject faults at all of these possible interfaces is able to test and verify the widest possible range of failure recovery mechanisms.

Other tools that are capable of providing a similar type of fault injection framework include the SAN-based Mendosus [9], as well as tools that modify a process's run-time image using the `ptrace()` function [3], or the `/proc` facility [6]. Generally, it has been our experience that `FIG` provides a more configurable and flexible interface than these tools. For example, the `FIG` configuration facilities (described below) allow any program run in a given session to be subjected to fault injection tests, while other tools often require that the targeted software module be explicitly run at the same time the fault injection tool is invoked.

A final note on related work is that the code instrumentation tool Xept [12] is capable of handling most, if not all of the error types that `FIG` can inject. Xept operates by instrumenting object code with extra routines that make sure all possible return conditions from library calls are handled. However, merely assuring the existence of such recovery code does not guarantee that it actually can offer recoverability from failures. Providing this guarantee is the primary goal of `FIG`.

2. IMPLEMENTATION

We hope `FIG` will prove useful in many different environments, so we made extensibility and flexibility a key component of its architecture. Our aim is for `FIG` to have low impact in terms of both its effect on the instrumented program (when not injecting faults) and the amount of effort needed to configure it to perform the desired fault measurements and injections.

2.1 FIG Architecture

Because `FIG` must instrument the interface between an application and the operating system, we need a method of inserting it between where the application makes a library function call and where the call is serviced by the OS. In the case of `glibc` calls, this point is the interface between the application program and the `glibc` shared library.

The UNIX dynamic loader provides a method for inserting code into this interface: by setting the LD_PRELOAD environment variable to point to the `libfig.so` library, we tell the loader to load it at the head of the library list. Symbol resolution proceeds from the head of the library list to the tail, so any symbol in an earlier library will override an identically named symbol in a later one. Calls defined within the `FIG` library, hereafter referred to as *stubs*, can thus override the `glibc` routines they intend to instrument. In addition to its simplicity, another benefit of this method is its ability to instrument any shared library, not just `glibc`.

Since the `FIG` stubs override the corresponding `glibc` symbols, we cannot make direct calls from `libfig.so` into `glibc` in order to execute the original library function. We use one of two methods to circumvent this limitation. Many symbols in `glibc` have *multiple definitions*: two symbols pointing to the same code. In this case, we call the secondary, non-overridden definition. For library functions without secondary definitions, we use the `dlsym()` routine to locate the function's code and call it indirectly through the returned function pointer.

2.2 Automatic Stub Generation

In order to facilitate our development effort, we created a mechanism for automatically creating stubs, which simplifies the addition of new instrumented functions to `FIG`. The developer adds some information, such as the function's name, secondary symbol, and parameter names and types, to a list of instrumented routines. Next, an AWK script generates the stubs' source code, which can then be built and linked into a customized `FIG` library.

Only a handful of calls cannot use this automatic mechanism. Some, like `mmap()` do not have secondary definitions, while others, like `malloc()`, interact with the `FIG` control logic and must be handled manually. Extending the stub generator to handle these cases would be straightforward, but may not be worthwhile if the number of such functions is small.

2.3 The FIG Interface

`FIG` consists of two major components: the `libfig.so` library and the `fig` command line interface. The `libfig.so` library implements system call interception and fault insertion as described above. The `fig` program sets up the environment as needed by `libfig.so` and then executes the program being tested. When the test program exits, `fig` prints out timing information, similar to the UNIX `time` utility.

The `fig` program also provides a convenient interface for passing configuration information to `libfig.so`. There are options to control the amount of logging performed, specify the location of output files, and enable or disable timestamps on log entries. The user can also specify a control file, which indicates which library calls will have faults inserted and controls the type and frequency of these faults.

`FIG` can be used without the `fig` program, which is actually little more than a wrapper that sets the appropriate environment variables for `libfig.so` but it provides an easier and faster interface than `libfig.so` alone.

2.4 Overhead

Since one intended use for `FIG` is performing fault insertion in running systems, it is vital to minimize its impact on the instrumented program's performance. Even during development, faster tools are generally appreciated. Our measurements of `FIG`'s overhead can be found in Table 1, which shows that `FIG` is quite lightweight and competitive with similar tools.

`FIG` provides a number of options for controlling the amount of logging it performs, which directly impacts its overhead. At the lowest logging level, where only injected faults are recorded, its overhead is very low: only 2.5% in our testing. Since a log of every library call made is of little value in a deployed system (not to mention a major use of disk space), we believe this level of overhead to be representative of the performance penalties that will be incurred when using `FIG` as an online fault injection tool.

Developers will likely appreciate a greater level of detail in the logs. `FIG` can log all instrumented `glibc` calls, or just those with errors (both naturally occurring and injected). With full logging enabled, the overhead increases to about 43%. Additionally, the user can add timestamps to each log entry, which roughly doubles the overhead at a given logging level.

Even with all logging and timestamp options enabled, `FIG` is still much lighter weight than the familiar `strace` tool. While not identical in function, `strace` records a similar variety of calls, but does not timestamp its log. It incurs over five times more overhead on our test program than `FIG` with comparable logging options.

3. TEST METHODOLOGY

Our initial tests of `FIG` consisted of using it to inject errors into several common UNIX programs. Our intent was to select a variety of programs, including simple command line utilities, typical desktop applications and supposedly reliable servers. All of these applications were tested on workstations running Red Hat Linux for Intel-based PCs, kernel version 2.4. We tested how each program handled faults from the `malloc()`, `read()`, and `write()` system calls. With some of the programs, we also tested the `open()`, `close()`, and `select()` calls when applicable. We chose these calls on the basis of earlier experiments with the UNIX `strace` tool, which indicated that these system calls are among the most frequently used by our test applications.

For each system call, we injected the types and frequencies of errors that we reasoned would likely be present in a system under stress from excessive load, hardware failures or software misbehavior. These included resource scarcity errors (ENOMEM – insufficient memory, ENOSPC – insufficient disk space), device failure errors (EIO – general I/O error) and errors that can occur during the normal execution of a program, but may be more prevalent when the system is heavily loaded (EINTR – system call was interrupted).

Our test applications:

1. *ls* — the common UNIX directory listing program.
2. *GNU Emacs* 20.7.1 — a large and mature text editor that can be run with or without an X Windows interface.
3. *Apache* 1.3.22 — an open-source HTTP server. We used a standard web browser to access a set of static

	User Time	System Time	Real Time	Overhead
without `FIG`	12.11	21.15	33.46	—
`FIG`: no logging	13.20	20.87	34.28	2.5%
`FIG`: logging without timestamps	24.29	23.02	47.83	42.9%
`FIG`: logging with timestamps	36.66	24.25	61.74	84.5%
`strace`	65.06	47.14	112.85	237.3%

Table 1: Timing using Berkeley DB to write and read one million words without transactions. The tests were run with and without `FIG` and `strace`. Using `FIG`, we enabled various options: suppressing log messages, logging but not timestamping each system call, and full logging and timestamping. All tests were done using five runs and averaging the results. All times are in seconds.

HTML pages through this server.

4. *Berkeley DB* 4.0.14 — an open-source flat file database library. Our test application was a port from TCL to C of one of the library's regression tests. This test loads an unsorted list of 10,000 words into the database and then reads them out in sorted order.

5. *Netscape Communicator* 4.76 — a popular web browser. Our tests involved loading and reloading a series of web pages from remote sites.

6. *MySQL Server* 3.23.36 — an open-source database server with full transactional properties and support for remote access. We used the accompanying MySQL client program to access, query and alter existing database tables that were stored on the server.

4. RESULTS

We now analyze the relevant results from testing the sample programs. A brief summary of our test results can be found in Table 2.

4.1 ls

We evaluated `ls` under a limited set of tests to see how a rudimentary UNIX application would fare under `FIG`. We had few expectations of proper recovery behavior on the part of `ls`, since it is not billed as being fault tolerant, but the results were a pleasant surprise.

When `ls` faces memory scarcity errors, it utilizes a rather interesting strategy: it retries the failing call but also keeps a global count of the number of `malloc()` failures that have occurred since the past successful call. Once the global count exceeds a threshold of five, the program exits with a warning to the user.

4.2 Emacs

We tested Emacs in two modes: with and without its X Windows interface (i.e., the `-nw` option), with the goal of isolating a potentially significant source of problems — namely, the windowing system — from the core program. A large part of X-enabled Emacs performs event management. In fact, under X, the single most frequent call into `glibc` is the `gettimeofday()` call, which is made nearly an order of magnitude more frequently than any other call. Thus, we surmised that differences in error handling may arise between versions of Emacs that do and do not use the X Windows system.

Both modes of Emacs are able to handle the EINTR error properly by retrying the failed call. However, under X, most other failures cause the program to detect an abnormal condition and then exit or cause a segmentation fault, indicating that the event processing loop does not perform well in the face of failures.

We also found that neither mode of Emacs is robust against memory exhaustion. The program is able to detect the first memory error, and displays a suitable message to the user: "Memory Exhausted – use M-x save some buffers RET." But if a subsequent memory failure occurs, Emacs crashes, leaving the user unable to save her work.

4.3 Netscape

Netscape exhibited the least recovery-oriented service model of all of the programs we tested. Regardless of the number of windows (navigator, email, etc.) open at the time, the program shuts down immediately and without any kind of warning to the user upon encountering an EIO or ENOSPC error. Usually this is a clean shutdown and not a crash—all preference and configuration files are closed before termination.

Perhaps the authors felt that this is reasonable behavior for Netscape, given that it is a client-side, nearly stateless application that makes no guarantees about reliability. However, Netscape's failure model raises questions about the type of failure-related user interface that even a non-robust application should provide. In particular, it seems reasonable to suggest that some sort of explanation for the program's peremptory termination should be provided to the user, so that he or she may be able to take steps to avoid encountering the same situation in the future.

Netscape does a better job of handling EINTR errors, usually just halting the current page load (sometimes with a warning dialog box) upon receiving EINTR. Again, considering Netscape's best-effort service model, it is acceptable to expect the user simply to use the "reload" button to resolve this situation.

4.4 MySQL

The MySQL server proved quite robust. Once the server is up and running, it provides a high degree of availability by using a pool of child processes to accept client connections and process queries. As we observed with injected `malloc()` failures, a child simply restarts when it encounters an error, relying on the remaining child processes to handle any incoming requests until it returns to full functionality.

In general, we found that the MySQL server aborts trans-

	`read()`		`write()`		`select()`	`malloc()`
	EINTR	EIO	ENOSPC	EIO	ENOMEM	ENOMEM
Emacs – no X	o.k.	**exit**	warn	warn	o.k.	**crash**
Emacs – X	o.k.	**crash**	o.k.	**crash**	**crash / exit**	**crash**
Netscape	warn	**exit**	exit	exit	n/a	exit
Berkeley DB – Xact	retry	detected	Xact abort	Xact abort	n/a	Xact abort
Berkeley DB – no Xact	retry	detected	**data loss**	**data loss**	n/a	detected, or **data loss**
MySQL	Xact abort	retry, warn	Xact abort	Xact abort	retry	restart process
Apache	o.k.	req dropped	req dropped	req dropped	o.k	n/a

Table 2: Results of testing Emacs, Netscape, Berkeley DB, MySQL and Apache with various failures. The programs and their configurations are listed along the left. Along the top, we list the different `glibc` calls which were tested and the particular error being injected. Boldfaced items indicate poor or undocumented behavior.

actions at the first sign of problems. Given the strict consistency requirements placed upon database servers, this behavior is acceptable because it prevents errors from corrupting the database.

4.5 Apache

Apache's error handling mechanisms were among the most robust of all of the applications tested. Like MySQL server, Apache utilizes the "process pool" approach to provide enhanced overall availability. In addition, the program avoids many resource exhaustion issues by allocating its entire internally managed memory pool during program startup and then not requesting more memory once it has reached a functional state. Thus, Apache does not suffer `malloc()` errors in the course of serving pages, since it never needs to call `malloc()`. However, our test involves serving only static pages, and one would expect pages utilizing CGI or Apache modules to require the allocation of additional memory.

Apache is also at an advantage with respect to writes to the file system. Apache only writes to the filesystem in order to maintain the log, a non-critical task. Thus, when presented with an ENOSPC error on a write (simulating a full disk), Apache simply ceases writing to the log, but continues serving pages.

Apache's handing of network I/O errors generally results in degraded service: most often, the request being handled is dropped. Apache successfully retries in the EINTR error case, but other errors cause the program simply to refuse to serve the requested page. This may in fact be the best policy in a networked environment, since network I/O errors often indicate network congestion or link failures, conditions that will not be improved by aggressive retries on the part of the server.

4.6 Berkeley DB

Finally, we looked at the Berkeley DB library. As can be expected, the database becomes corrupted when transactions are not used and environmental errors occur on write calls. Using transactions solves these problems, as the transaction can be cleanly rolled back without adversely affecting previously committed data. Thus, as pointed out by the Berkeley DB documentation, one should use non-transactional mode for data that is not important or that can be easily regenerated.

5. DISCUSSION & ANALYSIS

We now present some of the lessons and design patterns that we learned from both the successful and unsuccessful aspects of the applications we tested. We were not expecting to learn so much from such a limited suite of tested programs; this seems to point to the utility of the `FIG` tool as an aid to the programmer.

The design patterns that we learned turn out to be rather simple to implement in many different applications. They generally do not induce much overhead, but their use can alleviate or eliminate many sources of errors and unrecoverable problems.

Our results seem to indicate that server-side applications usually are engineered to be more robust than client-side applications, with more strictly defined interfaces and built-in contingency handlers. It seems likely that server applications are designed more carefully due to the higher expectations in terms of reliability and availability that are placed upon them.

In the following section, we also consider the possible uses of a fault injection tool like `FIG`. These include use as a software development tool as well as online testing of systems that have been developed to incorporate advanced failure recovery mechanisms.

5.1 Resource Preallocation

Dealing with scarce resources in the course of computing can be a challenging task. When faced with a failed `malloc()` call, we have seen that few applications check for the condition, much less recover from it. Given that applications should be checking for these cases of insufficient resources, what are they to do when they detect them?

One useful strategy is using resource preallocation to eliminate the possibility of scarcity-related failures during the life cycle of a program. We see this design pattern in Apache. It allocates all of its needed memory in the startup code of the initial process; no subsequent memory allocations are performed after this initial series.

Using this model, we see that Apache cannot encounter a `malloc()` failure in the middle of its processing, by virtue of its refusal to make any further calls to `malloc()`. We can apply this simple strategy of isolating resource acquisition in the initial startup code to other situations as well. For example, it is often useful at startup to open files that are known to be needed in the future.

The penalty for such a scheme is that resources may be over-provisioned at the time of application initialization. This owes to the fact that the actual resource requirements may not be known until "run-time" — i.e., until the resource must be used. The overhead of provisioning resources for times of trouble, however, can avert a catastrophic outcome and will reduce the number of crashes that a program encounters.

Resource preallocation does not fully solve the problem, however. The resource may be scarce at the time of program initialization, thus causing an error at that point. In fact, since we may be over-provisioning the resource, we are increasing the chance that we will encounter such a state of resource exhaustion. However, we posit that encountering a failure at program initialization is more desirable than doing so in the middle of program execution, when it is more likely that the state of the system will be disrupted by an abnormal termination of the operation in progress.

In many situations, full resource preallocation is not feasible. However, a limited form of this technique could alleviate many detrimental effects with a minimal cost. For example, one could imagine text processing applications that allocate a small amount of extra space, which allows them to properly save existing open files when free external memory is scarce. This technique limits overhead, while preventing the full catastrophe of lost data.

5.2 Graceful Degradation

We refer to techniques that offer at least partial service in the face of failures as providing "graceful degradation" of service. Such techniques, whereby errors lead to reduced functionality rather than outright failures, ameliorate downtime until an operator can fully recover the system. Graceful degradation is a central theme in the fault-tolerant computing philosophy.

One concrete example that we can point to is Apache. When the Apache log file is not able to be written, the rest of the service continues without interruption, but the log file is not written.

5.3 Selective Retry

Retrying errors seems to be a natural solution to resource exhaustion. Waiting a (tuneable) amount of time and retrying a failed system call can help in cases where resources later become available. `ls` uses this technique to retry failed calls to `malloc()`. Instead of retrying indefinitely, it maintains a global count of the number of `malloc()` failures encountered. Thus, it bounds the time to failure so that other measures can be taken if memory is not available. By comparison, Berkeley DB did not retry, but relied on the host application to do so; it returned the underlying error as a return code, allowing the caller to retry if it desired.

5.4 Process Pools

Both Apache and MySQL fork a pool of child processes at startup. The process pool approach provides a higher degree of perceived availability for the program, because the other child processes can mask the failure of a sibling by redistributing the processing load among themselves until the failed child can restart.

It should be noted, however, that while this approach certainly yields greater recoverability from localized failures, it may not be sufficient to keep the application running under widespread resource scarcity. In fact, such behavior may actually prove detrimental to the system, if continuous failed restarts by the child processes end up wasting even more system resources.

5.5 Potential Uses of FIG

At this point, we would like to speculate on some valuable uses of `FIG`. We propose that using fault injection in *production systems* can lead to better long-term reliability. It is certainly true that this radical view may not improve the robustness of systems built under current software methodologies. However, systems designed with advanced recovery mechanisms like reboot protocols, self-healing configuration support or fault-tolerant behavior should be able to take advantage of the ability of online fault injection to expose latent errors and software bugs.

`FIG` is also extremely applicable to contemporary applications such as Internet services. Using `FIG` to test programs in a development environment will help ensure the correctness of recovery-related code, while constantly encouraging developers to remain aware of the types of failures their programs may encounter.

To summarize, simply ignoring the problem of environmental errors and hoping that they will not occur is a perilous gamble. Since these types of errors *do* occur, it is important to ensure proper handling of them. `FIG` provides a straightforward mechanism for verifying that error-handling mechanisms function properly.

6. FURTHER WORK

Enhancements to `FIG` would allow greater testing flexibility. One area in which the enhancement would be particularly beneficial is that of the `FIG` control language. Allowing for clock time-based events would help to create failure scenarios wherein multiple calls fail at nearly the same time (i.e., `malloc()`, `read()`, and `write()` failures). This would allow one to more accurately simulate a severe resource shortage. Adding the ability for the language to inject failures only when a given condition holds would enable the simulation of faults in particular subsystems, such as faults on a given hard disk. Along these lines, supporting a Markov error model would add flexibility in being able to test varied error patterns.

It would be beneficial for `FIG` to grow to be a part of a larger suite of tools that can all be used toward improving reliability. One such combination would be to pair FIG with a root cause analysis system to allow for the diagnosis of unhandled errors.

In the larger view, it is evident that `FIG`, while useful in a wide variety of environments for numerous purposes, is still intrinsically limited by its dependence on the UNIX platform and its restriction to interactions at the application/library boundary. A possible long-term future project would be to investigate other abstraction levels for system instrumentation that are applicable to a larger variety of platforms and environments. A prime candidate for such research is the machine monitor layer of a virtualized execution environment.

7. CONCLUSION

We have seen how FIG can be used to find bugs in existing, mature software programs, and that even mature, reliable programs have poorly-documented interfaces and insufficient error recovery mechanisms. From this, we conclude that application development can benefit a great deal from a comprehensive testing strategy that includes mechanisms to introduce errors from the system environment. FIG fulfills this need by providing a straightforward method for introducing environment errors, which can then exercise seldom-used recovery code. FIG also allows us to evaluate both successful and unsuccessful application programming practices.

The short-term gain for an application in development is quite clear: running FIG during development will uncover improper error handling quickly, especially if high error injection rates are used. Running an application in production with FIG can help expose latent errors, but requires that the application already has a considerable level of advanced error detection and recovery mechanisms in place.

8. ACKNOWLEDGMENTS

We would like to thank Aaron Brown for his helpful suggestions and guidance throughout the project. Many thanks also to Jim Whittaker, the other members of the ROC team, and ROC seminar guest speakers Brendan Murphy and Jim Gray for their input.

9. ADDITIONAL AUTHORS

Additional authors: Dave Patterson (University of California, Berkeley, email: `patterson@cs.berkeley.edu`).

10. REFERENCES

[1] A. B. Brown and D. A. Patterson. Embracing Failure: A Case for Recovery-Oriented Computing (ROC). In *High Performance Transaction Processing Symposium*, October 2001.

[2] J. Carreira, H. Madeira, and J. G. Silva. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, 1998.

[3] M. Coleman. SUBTERFUGUE: A Framework for Observing and Playing with the Reality of Software. `http://subterfugue.org`, 2002.

[4] IBM Corporation. Autonomic Computing. `http://www.research.ibm.com/autonomic/`, 2001.

[5] Center for Software Engineering Research. Holodeck. `http://se.fit.edu/holodeck/`, 2002.

[6] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *6th Usenix Security Symposium*, 1996.

[7] Hewlett-Packard. SoS - Self-Organizing Services. `http://www.hpl.hp.com/research/itc/csl/pss/sos`, 2002.

[8] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek. Automated Robustness Testing of Off-the-Shelf Software Components. In *28th International Symposium on Fault-Tolerant Computing*, pages 464–468, 1998.

[9] X. Li, R. Martin, K. Nagaraja, T. Nguyen, and B. Zhang. Mendosus: A SAN-Based Fault-Injection Test-Bed for the Construction of Highly Available Network Services. In *1st Workshop on Novel Uses of System Area Networks (SAN-1)*, 2002.

[10] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[11] C. Perrow. *Normal Accidents.* Princeton University Press, 1999.

[12] K. Vo, Y. Wang, P. E. Chung, and Y. Huang. Xept: A Software Instrumentation Method for Exception Handling. In *The Eighth International Symposium on Software Reliability Engineering*, pages 60–69, November 1997.

[13] J. A. Whittaker. Software's Invisible Users. *IEEE Software*, pages 84–88, May/June 2001.