Compiler-directed Program-fault Coverage for Highly Available Java Internet Services

Chen Fu, Richard P. Martin, Kiran Nagaraja
Thu D. Nguyen, Barbara G. Ryder
{*chenfu, knagaraj,rmartin, tdnguyen, ryder*}@cs.rutgers.edu
Department of Computer Science
Rutgers University, Piscataway, NJ 08854

David Wonnacott
davew@cs.haverford.edu
Department of Computer Science
Haverford College
Haverford, PA 19041

Rutgers Department of Computer Science Technical Report
DCS-TR-518

January, 2003

Abstract

We present a new approach that uses compiler-directed fault-injection for coverage testing of recovery code in Internet services, to evaluate their robustness to operating system and I/O hardware faults. We define a set of *program-fault coverage metrics* that enable quantification of Java `catch` blocks exercised during fault-injection experiments. We use compiler analyses to instrument application code in two ways: to direct fault injection to occur at appropriate points during execution, and to measure the resulting coverage. As a proof of concept for these ideas, we have applied our techniques manually to *Muffin*, a proxy server; we obtained a high degree of coverage of `catch` blocks, with on average 85% of the expected faults per `catch` being experienced as caught exceptions.

1 Introduction

Internet services are increasingly becoming a part of our everyday life because of their promise to provide information anywhere, at any time. A number of researchers have recently observed, however, that current Internet services typically only achieve from 99% to 99.9% availability [Gra, ABB+02]. In contrast, the public telephone system, a system that we assume works anywhere, at any time, achieves around 99.999% availability [Gra]. This difference is highlighted by the prominence of news articles about failures of Internet services [Ele99, Reu99, Sil00, Swe00].

Currently Internet services are often constructed from Commercial Off The Shelf (COTS) software to meet short time deadlines and cost constraints. These COTS components may have been designed and implemented by diverse organizations using different coding and testing standards and methodologies. Therefore, ensuring the availability of such codes becomes the responsibility of the organization using them to build services. To exacerbate the problem, current tools are typically inadequate for testing fault detection and recovery code. Thus, the software is typically tested for functionality and performance, rather than availability, resulting in the current state of high unavailability.

In this paper, we argue that compiler analysis of application source or bytecode provides a powerful tool that can be applied to the problem of increasing the availability of Internet services. In particular, we explore a systematic technique for using compiler analyses to direct fault injection and measure the resulting coverage of recovery code. Our technique is designed to help testers identify faults to which the software is vulnerable[1], identify the location of the vulnerabilities in the code, and observe how the software handles the faults when they are injected to test vulnerabilities. Our approach is motivated, in part, by the observation that infrequently executed code exhibits a higher failure rate than frequently executed code [HC94].

We focus on analyzing the ability of software to handle hardware and operating system faults; we leave the testing of functionality vs. requirements to traditional testing techniques. We concentrate on I/O hardware faults since they are much more common than CPU or memory faults [TP99, KKI99]. We also focus on resource exhaustion faults and faults due to corruption of operating system data structures by bugs in the operating system. Our approach can be applied to software components as well as entire programs.

While many different approaches to fault injection have been developed and studied [CCH+99, DJM96, HSR95, KKA92, SVS+88], in a software engineering context all these efforts suffer from a fundamental limitation. Specifically, they have led to a probabilistic analysis that describes the likelihood that a program or software component can deliver correct service under specific fault and work loads [ACC+93], treating the application as a *black box* which only can be tested in terms of its observable behavior in response to inputs. While this probabilistic reasoning is necessary to produce dependable software, it is not sufficient for software designers and testers to understand how programming constructs, such as methods and statements, are affected by faults, nor does it ensure exercise of recovery code. For example, a tester may want to know how many different operations in the code can be affected by the same fault and if all these operations have been exercised through testing. When performing a fault-injection experiment, the system should allow the tester to know if a fault triggered an error and consequently the execution of specific error detection and handling code. In particular, if the program reads the disk in many different places, how can the tester identify all of these vulnerable operations and test them against appropriate disk faults?

We address this issue by using compiler analyses to identify code blocks that are vulnerable to faults, inserting instrumentation that directs the fault-injection infrastructure to inject the appropriate faults at these vulnerable operations, and tabulating coverage according to a metric that will be introduced in Section 2. In essence, we propose the adoption of what the software engineering community calls a *white box* testing approach [Bin99, Mar95, Mye79], where we use the compiler to look *inside* of software components to

[1] Software is *vulnerable* to a fault if it performs some operation that can trigger the fault, leading to an error. For example, an application that uses the network, but does not use the disk, is vulnerable to network faults, but not disk faults.

help the tester use a fault-injection infrastructure to maximal effect. This is similar to a number of software engineering approaches that examine the code to measure test coverage in terms of program constructs such as branches, statements and definitions-uses of variables [RW85, Bin99, Mar95, Mye79]; these measures try to quantify how many of these application constructs have been exercised by the testing process. In our case, we will concentrate on services (and/or components) developed in Java, and so the program constructs of interest are `try/catch` blocks.

We aim our work at Java-based services for many reasons. First, unlike C where programming convention often overloads the return mechanism to describe errors, Java contains well-defined program-level constructs, *exceptions*, that respond to error conditions [AG97]. This facilitates both the construction and analysis of error recovery. Second, a Java program which may experience fault-induced errors cannot be written without inclusion of the appropriate code to handle exceptional conditions. Third, Java is used increasingly in building dependable systems, in particular in the context of large-scale servers. Finally, the platform independence of Java, its portable program representation, (i.e., bytecode), and its defined JDK [SM] libraries make software reuse via COTS components easier and more desirable than for languages such as C and C^{++}.

Contributions: In this paper, we define the problem of testing Java-based Internet services to improve availability, using a white box technique. We present our advances over the current state-of-the-art below.

- We explore the connection between fault injection and coverage of program-recovery code in a layered system, defining the problems that must be addressed;
- We define a *white box* coverage metric for testing fault-recovery code in Java applications;
- We describe compile-time techniques to automatically instrument Java code to
  - direct a fault-injection infrastructure to inject faults at appropriate points in the execution of the program to exercise specific pieces of fault-recovery code, and
  - measure the faults and corresponding recovery code covered by a given test;

  This work includes the definition of an API for communication between the compiler-inserted instrumentation and a fault-injection engine;
- We present a feasibility study in which we have manually instrumented a sample benchmark to test for recovery from a set of faults. We achieve 100% coverage of four of the seven `catch` blocks where faults were injected, with an on average fraction per `catch` of approximately 85% of the expected faults actually being experienced as caught exceptions.

Overview: In Section 2, we present several possible white box definitions of fault coverage and give our reasons for selecting the metric we use in this work. In Section 3 we present our analyses used to instrument

Java applications to measure coverage and direct fault injection. We describe the API we have defined for communication between the compiler and Mendosus [LMN+02], the specific fault-injection engine that we use for this study. In Section 4, we give the results of our initial test of our approach on a single benchmark, *Muffin*, a proxy server. Our study was accomplished through manual application of our compiler algorithms and use of the modified version of Mendosus. Finally, in Sections 5 and 6, we discuss related work and give our conclusions.

2 Defining Coverage for Fault-recovery Code in Java Programs

Before giving our definition of coverage for fault-recovery code of Java programs, we first review prior uses of the term *coverage* and discuss the relation between operating system/hardware faults, Java exceptions, and exception handlers in the application. After this background, we discuss possible coverage metric choices and give our reasons for selecting the one we use in our experiments. We conclude this section with a discussion of some issues related to the measurement of coverage.

2.1 Comparing Definitions of Coverage

Both the dependability and software engineering communities have precise definitions for the term *coverage*; however, they use this term in very different ways. In the dependability context, coverage is defined as the conditional probability that the system properly processes a fault, given that the specific fault occurs [BCS69]. Later work included the assumption that the fault was *activated* in the probabilistic definition [ACC+93]. A number of modeling and analysis strategies naturally arise from this definition. First, coverage can be mathematically represented as *probability density* and *cumulative density functions* (PDF and CDFs). Second, these functions can be transformed into probability density over time and cumulative density over time, leading to a range of analyses using stochastic process models (e.g., [DT89]). These models can describe the impact that coverage has on the expected time to enter a failure state under a given fault load, and the amount of redundancy necessary to achieve targeted levels of availability and performance.

By contrast, the software engineering community uses a fundamentally different definition of coverage. In this context, coverage is defined as the fraction of the application that has been exercised by a given test in terms of specific programming constructs including statements and branches. For example, *all-branch* coverage ensures that every branch in a program (e.g., exits from an `if` statement) is traversed at least once during testing. Similarly, *all-statement* coverage guarantees that every statement in the program has been executed at least once during testing. Another set of constructs, based on dataflow, traces values from their definition point to their subsequent usage, that is, *def-use* coverage [RW85]. The *all-defs* coverage metric requires that tests cover one path between each value-setting operation and a use of that value [RW85]; this is to ensure that errors due to incorrect flow of data values are handled properly. The all-defs metric is the traditional dataflow metric most closely related to the new metric we define for fault coverage in Section 2.3. A hierarchy of def-use coverage metrics has been defined; these vary in power, in the sense that one has more confidence in the correctness of a program tested using a higher coverage metric than a lower one.

For the remainder of this paper, we call the definitions based on conditional probability *fault coverage* and the software engineering definitions, *program coverage*. One of our primary goals in this work is to define a metric for program coverage as it relates to faults and fault-recovery code. This will be called a *program-fault coverage* metric because it measures the fraction of the program run in response to a fault load. Some of the program-fault metrics we define for complete applications (see Section 2.4) are reminiscent of the conditional probability definitions used in the dependability community, but our metrics describe the coverage of combinations of recovery-code blocks and fault types, not the fraction of actual faults that were handled.

2.2 Relating Faults to Exceptions

The software engineering program-coverage metrics are motivated by a desire to know what parts of the application have been explored by the testing process. Since we are measuring the response of a Java application to faults that are returned by the operating system or I/O hardware (e.g., disk errors or bad data from the network), we focus our attention on Java exception handling code. The challenge is to map lower-level faults to program-level exceptions and find their corresponding program-level exception handlers. In the rest of this paper we use the terms *exception* and *exception handler* to refer to these program-level constructs.

In order to explore the relations between faults, exceptions and exception handlers, we present the following simplified discussion of Java exception handling. For details, see [AG97]. In Java, operations and method calls generally indicate the presence of errors by throwing an exception instead of returning a value. Code that can throw an exception may be enclosed within a `try` block with one or more associated `catch` blocks, each of which identifies the type of exception it can handle. If an operation in the `try` block throws an exception, program control is transferred directly to a `catch` block with matching exception type. Java's subtyping rules can be used to write `catch` blocks that will be triggered by multiple types of exceptions; a `catch (Exception e)` matches any type of thrown exception.

For example, a programmer may enclose a sequence of operations that read from a file within a `try` block that has an associated `catch` of `IOException` (or a more general `catch` of `Exception`) containing code to recover from file read errors. If any of the read operations encounters an error and throws `IOException`, the program will go directly to that `catch` block.

Exception handling code may be located either in the method performing a vulnerable operation or in some method which directly (or indirectly) calls this method. When an exception is thrown, the JVM searches for an appropriate `catch`, beginning in the method performing the throw, and, if necessary, working "backwards" to a caller method containing the handling code. All of the exceptions we discuss are classified by Java as *checked* exceptions, meaning that methods which contain vulnerable operations that may trigger these exceptions, need either to handle them or to pass them explicitly "backwards" to a caller to handle [AG97][2].

A Java operation may be vulnerable to a variety of faults, and the exception generated may depend on both the fault and the operation. For example, `reads` and `writes` exposed to faults that produce the operating system error code **NET_EAGAIN** may cause different exceptions (namely, `IOException` and `SocketException`). In contrast, the error codes **NET_EPIPE** and **NET_EFAULT** can both result in `SocketException`. (See Section 4 for more details on specific faults.) So the relation between faults and exceptions can be one-to-many or many-to-one. There is generally a unique Java exception for any specific fault-operation pair, but our approach does not rely on this fact.

Our techniques make use of a table mapping fault-operation pairs to (one or more) exceptions. Unfortunately, the construction and use of this table are both complicated by the layers of software between the hardware and the application being tested (such as device drivers, the operating system, and the Java Virtual Machine (JVM)). These layers can have a dramatic impact the way in which low-level faults translate into exceptions at the program level.

In the simplest case, a fault can cause an immediate exception, and thus direct control to the appropriate `catch` block for a vulnerable operation that was executed while the fault was activated.

In some cases, recovery strategies at a lower level of the system can entirely prevent a higher-level layer from observing a fault activation. For example, at the operating system level, the time out and retry of a TCP socket can easily mask transient hardware link errors, and thus no application-level `SocketException` is ever thrown. Likewise, a mirrored file system can hide many types of SCSI disk errors from the JVM. This effect must be considered in the construction of our mapping of fault-operation pairs to exceptions, and our coverage metric (defined in Section 2.3) accounts for this kind of lower-level fault handling.

Additionally, layering may cause *latent errors*. For example, input buffering can allow numerous disk

[2] Java also has *unchecked exceptions* such as **NullPointerException** that do not need explicit handling.

reads to succeed after a fault, until all buffered data have been consumed; at that point, if the fault is still active, a later `read` may throw an exception. Likewise, the JVM may not observe socket exceptions for minutes, even in the face of total link failure, because TCP attempts numerous retries in the face of lost packets. In this case, data that was written during the time of the fault may be lost, without causing any exception (there may not be any more I/O operations by the time TCP gives up). The potential for latent errors influences the interpretation of our run-time collection of coverage data, in that we cannot assume that exceptions caused by a fault that occurs during the execution of a given `try` block will necessarily appear in the expected `catch` of that particular `try`.

2.3 Fault-Catch Coverage

Given the complex relationship between faults and exceptions, we envision that any given testing effort would begin by defining an explicit universe of faults to be studied, which we call F. Each `catch` block in the program can potentially be triggered by some subset[3] of F that we call f. On a given test run, some subset of f, which we call e, will actually trigger this `catch`.

The program-fault coverage metric we have chosen for our work is somewhat analogous to the *all-defs* metric, but with faults playing the role of potentially defining error states, and `catch` blocks "using" the error states to perform a recovery action. We first define a metric for a single `catch`, and leave discussion of metrics for larger regions of code for Section 2.4.

Definition (Fault-catch Coverage Metric): Given a single `catch` block c that can potentially be triggered by a subset f of the fault universe F, and a set of test runs t in which fault set $e \subseteq f$ trigger exceptions that reach c, the *Fault-catch Coverage* of c by t is $\frac{|e|}{|f|}$.

We chose this definition over several other possibilities for a variety of reasons. Perhaps the simplest definition of coverage would be to see if all `catch` blocks, or all statements of all `catch` blocks, have been executed by a test; these definitions are basically analogous to the traditional *all-statement* program-coverage metric. However, a single `catch` can be triggered by many different faults or exceptions, and under this definition we might obtain a high coverage measure without actually testing the response of the application to a large set of faults.

This same problem of failing to test the response to a particular fault could occur if we measure the

[3]This subset will exclude any faults that are not related to the operations in the associated `try` (i.e. the subset for a `catch` corresponding to a `try` with no I/O operations would not include `IOException`). Furthermore, we also exclude faults that will be handled by lower layers of software (or hardware). The tester may also choose to exclude faults that are not relevant due to program usage.

coverage of pairings of exceptions and `catch` blocks. Furthermore, if the code must respond differently to various exceptions, the programmer will typically write several `catch` blocks, and our definition will identify some of these as not covered if the corresponding exceptions have not been raised.

We could also define coverage in terms of pairings of `catch` blocks and program points at which faults (or exceptions) can occur. Under this definition, a `catch` block following a `try` block containing eight statements that read from a file would need to be executed at least eight times (or, perhaps, $8f$ times) to be considered fully tested. If we require each statement in the `try` to be tested for each of the f faults, this definition is stricter than ours (in that the software must be tested at least as much to be considered fully tested). However, we believe it would focus attention on issues that are typically irrelevant (i.e., testing a fault in each `read` statement) rather than fundamental issues (i.e., testing each kind of fault).

There are several intermediate points between our fault-catch definition and the extremely strong definition suggested above. We could require a test involving each fault and a test involving each source statement ($max(8, \mid f \mid)$ tests in the previous example). Another possible middle ground is an object-oriented definition: require a test for each fault for each object that could be involved (if three of the eight reads use one file, and the remaining five a second file, this would require $2 \mid f \mid$ tests). We leave investigation of these options for future work.

In all of these discussions, our knowledge of which exceptions can possibly be handled by which `catch` blocks is derived from a static representation of the Java source or bytecodes, rather than data from the executing program. Therefore, we are making the usual assumption of compile-time program analysis, namely, that every static path in the program representation is actually executable. This may not be the case, so that if we use a def-use type coverage metric from exception occurrence site to `catch` block, then we may include some infeasible def-use relations, which can never be covered. This is a common problem in software testing as well; it is addressed by using as precise as possible program analysis to eliminate infeasible paths where possible and by human examination.

2.4 Aggregating and Reporting Coverage Results

There are several ways to produce aggregate information about code that contains many `catch` blocks, such as an entire application, a library, or a new unit of code that is being added to a working application. Consider code with n `catch` blocks $c_1 ... c_n$, in which c_i can be triggered by fault set f_i, and a test t in which faults in set e_i have each produced an exception that reaches c_i.

Definition (Average Fault-catch Coverage:): The average of all the ratios $\frac{|e_i|}{|f_i|}$.

Definition (Overall Fault-catch Coverage:): The ratio of the total numbers of tested and possible faults, $\frac{\sum_{i=1}^{n}|e_i|}{\sum_{i=1}^{n}|f_i|}$

Definition (Fraction of Covered Catches): The fraction of the catches for which $\mid e_i \mid = \mid f_i \mid$.

We leave as an open question under what circumstances different aggregate measures are best, because we strongly suspect that no single aggregate will capture all user needs. We therefore envision the use of a language-aware software tool (such as the Eclipse IDE [ecl]) that could maintain the raw data about vulnerable operations, faults injected, thrown exceptions and covered catches. This tool could then present whatever metric is chosen by the tester, or even help the tester identify inadequately tested `catch` blocks or faults for which little testing has been done.

2.5 Measuring Fault-catch Coverage

The set of faults that may be associated with each `catch` does not vary from test to test, and can thus be computed statically. Unfortunately, an analysis based on the type of exception declared in the `catch` could produce a dramatic overestimate of f_i for many catches, since the declared type may be a supertype that subsumes many exceptions that cannot actually be thrown. This same effect applies to the exception types declared by methods called in the `try` block. Thus, to overestimate as little as possible, we perform an interprocedural analysis of the code in the `try` block. Intuitively, using the calling structure of the program, we find a primitive operation that actually throws an exception and then propagate it backwards on the calling structure to find its list of callers, stopping at the "nearest" `try` block. Details of this analysis are left for Section 3.

Information about the faults that actually trigger each `catch` must be collected separately for each run. We do this by instrumenting each `catch` block to record its identifier, the class of exception that reached it, and the fault associated with the exception. We do not currently record the source (e.g., `throw`) of the exception, but we could in principle do so using the JDK method `Throwable.printStackTrace()`.

Note that our current experimental system simply records the fault that is currently being injected. This requires communication with the fault-injection engine (see Section 3.2), and is most easily accomplished if there is never more than one (simultaneous) fault injected (as in the experiment in Section 4). Furthermore, our current system cannot actually guarantee that the injected fault caused the exception. For example, if we inject a disk fault in one block, but the disk fails (or produces an `IOException`) in some other way, we will record that the injected fault reached the `catch` block. If this proves to be a problem, we will explore systems for tracking information about injected faults across the program/system boundary.

3 Injecting Faults to Improve Coverage

We now consider how the compiler can instrument application code to communicate with a fault-injection engine at run-time, in order to direct the fault-injection process to obtain high program-fault coverage as measured by our metric. Specifically, we use Mendosusas our fault-injection infrastructure, but our approach is generally applicable to a wide variety of fault-injection systems. We choose Mendosusbecause it can produce error conditions returned by the operating system as well as I/O hardware faults. Mendosuscan also emulate different network configurations (i.e., different target platforms) and generate system-wide faults such as a switch failure.

For this work, we have extended Mendosuswith an API for dynamic external direction as to when specific faults should be injected. Previously, Mendosusinjected faults according to a pre-determined script comprised of traces and/or random distributions. Our basic approach currently is to identify a statement inside a `try` block, where the software has committed to the execution of some vulnerable operation (such as a `read` from a file), but before the operation itself is performed. At this program point, we insert instrumentation to select an appropriate fault and to direct Mendosusto inject this fault, using the API described in Section 3.2. Once execution reaches the corresponding `catch` block, or the end of the `try` block, we direct Mendosusto cancel the injected fault.

In many cases, our fault-injection instrumentation can be placed immediately before the vulnerable operation, just as a fault could be emulated by throwing an exception in (or instead of) the operation itself. In general, however, Mendosusmay require advanced warning of the fault to be injected, in case it impacts multiple nodes in a distributed application. For this reason, we move the instrumentation backward in the code as far as we can, possibly all the way to the beginning of the `try` block, although no farther because we want to plant the fault only when it has a chance of exercising the specific `catch` block of interest to obtain coverage. In the future, we may investigate the use of profiling techniques to provide an estimate, for specific program points, of the amount of time until the vulnerable operation will be triggered; it is not clear that accurate timing information will be needed.

3.1 Compiler Analyses

Two dataflow analyses allow us to accomplish both the communication with Mendosusand the recording of the fault-catch coverage achieved. Both are performed on Java bytecodes, so we can apply them whether or not source code is available. The first analysis, *exception handler analysis*, essentially traces backwards from an excepting operation (or call) in the Java code to its handler which will be on the call stack when the exception occurs. The second analysis, *resource points-to analysis*, finds all objects reachable from the (fields of) actual arguments in a method call; this analysis is necessary to give access to objects such as file descriptors, which may result in excepting computations during the lifetime of this method call.

Exception handler analysis uses a compile-time representation of the program *call tree* [Set96] to guide the backwards search from an exception occurrence point to a handler. The *call tree* records the sequence of method calls that may occur during execution in a tree structure. Its nodes are methods and its edges connect the calling method with the called method (annotated by the call site). The call tree can be approximated by compile-time class analysis [TP00, BS96, DGC95, GC01] or reference points-to analysis [RMR01, LPH01, MRR02]. The exception occurrence point may be either a Java library call whose JNI routines generate the exception or a specific Java method call which throws the exception (and does not handle it). By searching backwards on the call tree, we can find the closest exception handler for the exception, according to Java exception semantics[AG97]. The backwards search on the call tree requires us to examine each method call to ascertain whether or not it is included in a `try` block that handles the exception type whose handler we seek.

Once we find the handler, the associated method call in the `try` block becomes the focus of our placement of communication with Mendosus; the type of fault requested depends on the operation(s) at the exception occurrence. In the actual implementation of our prototype, we will use an approximation of the call tree, a potentially exponential-sized structure; possible choices to be investigated include a calling context tree [AS00] or a call graph with annotations about call site locations within its nodes (i.e., methods). We plan to experiment with these different program representations in order to balance analysis cost with accuracy.

Resource points-to analysis allows us to find the specific object on which the excepting computation occurs; this is necessary to determine the set of possible faults to be injected. *Points-to analysis* enables approximation at compile-time of the set of objects to which some reference variable can point at run-time. When the solutions at distinct method call sites are differentiated by the analysis so that different points-to information can be associated with them, then the analysis is termed *context-sensitive*. We will use a context-sensitive reference points-to analysis to ascertain those objects necessary for the vulnerable operation, even if references to them are stored in fields of other objects. We need the type of the object to select appropriate faults to inject; however, it may not be possible to determine the appropriate set of faults to inject until run-time, because they are determined by the run-time type of an object. For example, an open `InputStream` may correspond to a `FileInputStream`, in which case disk faults are appropriate, or it may correspond to an input stream from a socket, in which case network faults are appropriate. We will use the reflection library in the JDK to determine these types at run-time. This library allows run-time examination of object properties such as type and value. We need values for some of the object's fields to provide to Mendosus(e.g., the file descriptor for an input stream).

These two analyses, described above, pinpoint the constructs in the application that we must instrument. The first identifies both the `try` blocks into which we insert fault-injection and cancellation code and the associated `catch` blocks that we instrument to measure coverage. The second analysis provides information about the objects involved, which is needed to select appropriate fault types and parameters for communicating with Mendosus, as well as information that is needed to analyze method calls in order to construct the call tree.

The above analyses do not address all latent errors – we plan to explore this important issue in future work.

3.2 An API for Instrumentation-Driven Fault Injection

In the instrumented application code, we need to inform Mendosusto inject a fault or to cancel a previously injected fault. The kind of fault determines the appropriate parameters needed. To facilitate the communication with Mendosus, we implemented a user-level client Java library exporting the following methods:

`public static boolean inject(int faultType, int interval, SomeList parameters)`

This method requests an injection of a fault of type `faultType`, which will expire after `interval` number of seconds. The `faultType` is determined using the run-time type of the object (e.g., file descriptor or a communication socket), as a key into a list of fault types provided by Mendosus. The parameter list `parameters` contains additional information to guide Mendosusin the injection of the fault,[4] such as the file descriptor. The boolean return value specifies whether Mendosussuccessfully injected the requested fault.

`public static boolean cancel(int faultType, int interval, SomeList parameters)`

This method asks Mendosusto cancel an on-going fault of type `faultType`. The boolean return value specifies whether Mendosuswas able to locate and cancel the fault.

Instrumented application code and Mendosuscurrently communicate synchronously: on successful return of the `inject` method, the fault has been injected, and any subsequent use of the affected resource

[4]The tuple (`faultType`, `interval`, `parameters`) serves as unique identifier of an injected fault in Mendosus.

(within `interval` seconds) will produce an error. Similarly, on a successful return of `cancel`, the previously injected fault already has been canceled. For our tests, described in Section 4, we used an `interval` large enough to ensure that faults remained in effect until explicitly canceled. Our preliminary experiment suggests that this synchronous approach is sufficient except in the presence of latent errors.

While our algorithms are not sufficient to handle the general problem of latent errors, we can use several simple variations on our approach to improve fault-catch coverage in the presence of latent errors. First, we can use the approach described above, which we label *fault-cancel mode*, though this may cause faults to be canceled before latent errors are observed. Second, we can use *fault-not-cancel mode*, in which we never cancel a fault once injected. Finally, we can use *fault-reinject* mode, in which faults are canceled, but then injected again at a later execution of the `try` (in a separate run of the program).

3.3 Testing Multi-Threaded and Distributed Applications

Internet services are generally built as multi-threaded or distributed applications. Several additional issues must be addressed when testing a multi-threaded application running one or multiple nodes.

For a multi-threaded application running on a single node, as is the case for our initial benchmark, the question arises: What happens when the thread that requested a fault injection is context-switched out before the fault has been canceled? Three scenarios are possible:

1. the fault does not affect other threads that run before the original thread is allowed to run again,
2. some other thread is affected by the fault and crashes the application, or
3. one or more other threads are affected by the fault, but they recover sufficiently to not crash the application, eventually allowing the original thread to run again with the fault still activated. If more than one thread executes the same `try` block and experiences the error caused by the injected fault, we will count the `catch` block as covered, regardless of which thread actually executed it.

The first case clearly does not raise any concern. The second case is a successful test that requires bug fixing in the application before this particular fault-catch pair can be exercised. In the third case, our current instrumentation will count the coverage of the fault-catch pair in the original thread, as expected, but not any other `catch` block that was exercised "incidentally". This is not a problem, as a failure to notice the incidental coverage will simply cause our system to perform an unnecessary test.

Testing of distributed applications running on multiple nodes raises at least two additional challenges. First, we may wish to inject a fault that affects several nodes as soon as one thread reaches a particular `try`. Our current system can insert instrumentation to do this, but we have not yet investigated how best to perform fault injection. For example, we may or may not wish to allow the thread injecting the fault to continue before the entire distributed fault injection is complete. If the thread is not blocked, we must make sure that the fault is injected far enough in advance of the vulnerable operation – we believe this goal will produce the greatest challenge for our instrumentation algorithms.

Second, the question of how to inject faults at the time when some system-wide condition has been achieved in a distributed application raises even more interesting issues. Simply blocking the thread that is communicating with Mendosusis not sufficient; other threads on other nodes may be progressing past the vulnerable point of interest. This has been partially studied before [CCH+99] although not in the context of compiler-directed fault injection. We leave this as an important issue for future work.

4 Feasibility Case Study

To offer proof-of-concept for our methodology we performed a small case study using a *http* proxy server called *Muffin* [muf]. In this experiment, we hand simulated our compiler-directed analyses to determine where to inject faults and inserted by hand instrumentation for communication with Mendosusand recording of coverage. We studied all faults using fault-cancel mode, and additionally used fault-not-cancel mode for latent errors; we have not yet implemented fault-reinject mode in our test system.

4.1 Muffin and its Possible Faults

Muffin is a single-node, multi-threaded application whose interactions with the operating system are mainly receiving and sending data over the network (i.e., relaying requests and web pages). The disk access is essentially to log fulfilled requests without ever trying to read them back in; this data is easily stored in cache, so the footprint is too small for meaningful experiments. Instead, we concentrated on introducing faults related to network I/O.

Table 1 gives the faults used in our study (i.e., our universe of faults F, as mentioned in Section 2.3). These faults are divided into two classes. The first class is network hardware failures such as NIC, link, and switch failures. For a single-node application, these all have the same effect (i.e., the inability to read or write data), so we experiment with only one entry from this class, namely **NIC_DOWN**. In addition, we do not consider transient packet loss because Muffin depends only on TCP, which completely hides such faults from the application unless its seriousness approaches that of a **NIC_DOWN**. The second class is made up of faults resulting from the operating system such as exhaustion of resources or a corruption of essential data structures. The complete set of network operations used in Muffin is: `bind, connect, accept, read, write`.

Table 1: Faults Used in the Experiment

Fault Class	*Fault Type*	*Operations*	*Description*
Hardware	NIC_DOWN	All except bind	Drop all the packets coming from or to a given IP for a certain amount of time to simulate some network hardware failure
OS error code	NET_EBADF	bind,read,write	Bad socket number
	NET_EFAULT	read,write	Buffer space unavailable
	NET_EPIPE	read,write	Socket with only one end open
	NET_EAGAIN	connect,read,write	Necessary resource temporarily unavailable
	NET_ENOMEM	bind,accept	Not enough system memory
	NET_ECONNREFUSED	connect	Connection refused (e.g., remote node crashed and not yet recovered)

Table 2: Vulnerable Operations in `try` Blocks

`try`	*Operations*	*Possible Faults*
0	read	NIC_DOWN, NET_EBADF, NET_EFAULT, NET_EPIPE, NET_EAGAIN
1	read/write	NIC_DOWN, NET_EBADF, NET_EFAULT, NET_EPIPE, NET_EAGAIN
2	write	NIC_DOWN, NET_EBADF, NET_EFAULT, NET_EPIPE, NET_EAGAIN
3	read	NIC_DOWN, NET_EBADF, NET_EFAULT, NET_EPIPE, NET_EAGAIN
6	accept	NIC_DOWN, NET_ENOMEM
7	connect	NIC_DOWN, NET_EAGAIN, NET_ECONNREFUSED
8	bind	NET_EBADF, NET_ENOMEM

In Muffin, we found seven `catch` blocks that can catch `IOExceptions` due to the faults in Table 1. Figure 1 shows a high-level view of the control flow in Muffin and the location of the `try` blocks associated with these `catches`. Each `try` block has only one associated `catch`; the numbers shown in Figure 1 serve as identifiers of each `try` block or the associated `catch`. We manually instrumented the `try` and `catch` blocks as per the algorithms of Section 3.

Table 2 lists the vulnerable operations in each `try` and the corresponding faults which can cause exceptions reaching the associated `catch`; in the terms defined in Section 2.3, these comprise the relevant fault sets f_i for each `catch` i. Note that `try` blocks 0-3 are all vulnerable to the same set of faults, as they all send or receive data, but different sets of faults may affect `try` blocks 6-8, which involve the establishment of network connections.

4.2 Experiment Specifics

In our experiment, the proxy server (Muffin version 0.9.3a), the actual http server (Apache), and a synthetic client that generates http requests are running separately, each on one of three 800 MHz PIII PCs under Linux 2.2.14-5.0. We used the IBM Java 2.13 Virtual Machine for Linux.

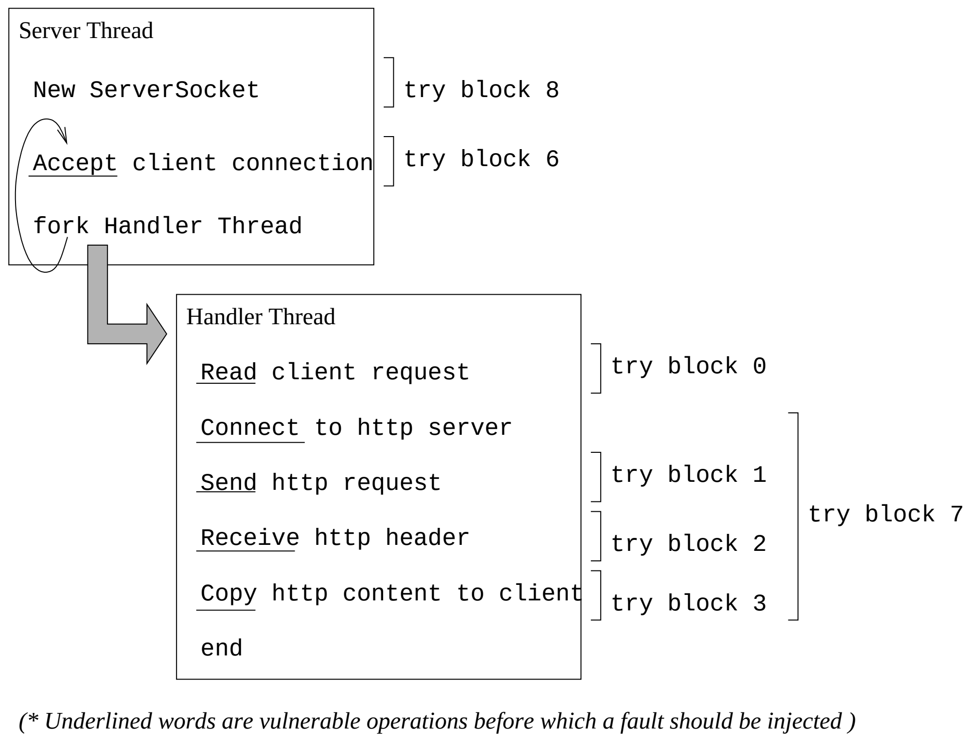

Figure 1: Structure of Muffin and selected `try` blocks

For each test run of Muffin, we injected a single fault into one instrumented `try` block in fault-cancel mode. One run was performed for each valid fault-`try` combination (see Table 2) and data recorded for all these test runs. Table 3 shows the results of our experiment. The *Faults* column gives the e_i sets discussed in Section 2.3.

In all the tests, all faults except **NIC_DOWN** are recorded in all appropriate `catch` blocks, showing that our methodology can drive the application through all of its responses to these faults, obtaining good test coverage for them. However, **NIC_DOWN** often causes latent errors, and its injection into the six vulnerable `try` blocks yielded only two covered `catches`. We re-ran our tests for **NIC_DOWN** in fault-not-cancel mode and were able to also cover `catch` 3 with this fault.

Figure 2 summarizes the $\frac{|e_i|}{|f_i|}$ values for each `catch` graphically, and Table 4 gives our aggregate coverage metrics for the tested code. Our (fraction of) *covered catches* metric is the most stringent, drawing attention to the fact that about half of the `catch`es have not been fully tested. The other two metrics take into account the amount of coverage of the partially covered `catch`es. In this experiment, we obtained slightly higher values for *overall fault-catch* coverage than *average fault-catch* coverage, as the former effectively weighs the individual catch average ratios by the number of associated faults and our lowest percentage

Table 3: Faults and Exceptions Recorded For Each Catch Block

Catch	*Faults*	*Exceptions*
0	NET_EBADF	java.net.SocketException: socket closed: Bad file descriptor
	NET_EFAULT	java.net.SocketException: Bad address: Bad address
	NET_EPIPE	java.net.SocketException: Broken pipe: Broken pipe
	NET_EAGAIN	java.net.SocketException: Resource temporarily unavailable
1	NIC_DOWN	java.io.InterruptedIOException: Read timed out
	NET_EBADF	java.net.SocketException: socket closed: Bad file descriptor
	NET_EFAULT	java.net.SocketException: Bad address: Bad address
	NET_EPIPE	java.net.SocketException: Broken pipe: Broken pipe
	NET_EAGAIN	java.net.SocketException: Resource temporarily unavailable
2	NET_EBAD	java.io.IOException: Bad file descriptor
	NET_EFAULT	java.io.IOException: Bad address
	NET_EPIPE	java.io.IOException: Broken pipe
	NET_EAGAIN	java.io.IOException: Resource temporarily unavailable
3	NET_EBADF	java.net.SocketException: socket closed: Bad file descriptor
	NET_EFAULT	java.net.SocketException: Bad address: Bad address
	NET_EPIPE	java.net.SocketException: Broken pipe: Broken pipe
	NET_EAGAIN	java.net.SocketException: Resource temporarily unavailable
6	NET_ENOMEM	java.net.SocketException: Cannot allocate memory
7	NIC_DOWN	java.net.NoRouteToHostException: Connection timed out
	NET_EAGAIN	java.net.SocketException: Resource temporarily unavailable
	NET_CONNREFUSED	java.net.ConnectException: Connection refused
8	NET_EBADF	java.net.SocketException: Bad file descriptor
	NET_ENOMEM	java.net.SocketException: Cannot allocate memory

Table 4: Aggregated Report of Coverage

	Average Fault-catch	**Overall Fault-catch**	**Covered Catches**
Cancel fault at end of try	84.3%	85.2%	42.9%
Without the cancel	87.1%	88.9%	57.1%

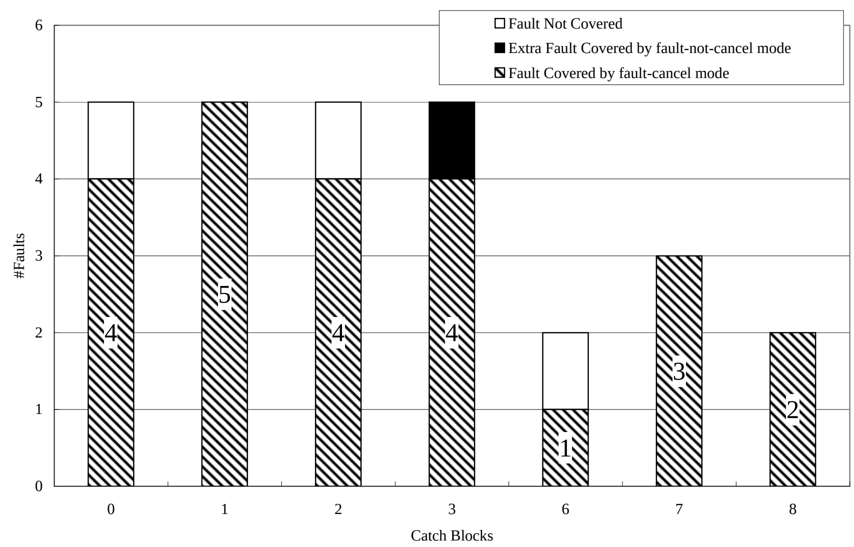

Figure 2: Coverage Data

coverage occurred on a `catch` with only two faults.

Our data show that we can inject faults, instrument programs to measure fault-catch coverage, and achieve significant levels of fault-catch coverage for Muffin. For faults that do not produce latent errors, we produced 100% fault-catch coverage, suggesting that our techniques are valuable. We were less successful with faults that do produce latent errors, covering four of the seven **NIC_DOWN**/`catch` combinations in fault-not-cancel mode. While these coverage results are valuable in that they guide the tester to those fault-recovery codes that are not fully tested, as discussed before it is very important to improve our coverage for these faults resulting in latent errors.

5 Related Work

Researchers in the dependability and software engineering communities have studied the problems of program coverage and fault coverage extensively. Given the limited space, we will focus here on a comparison of our work with previous research on fault injection using program-coverage metrics. An understanding of probabilistic fault coverage [BCS69], its relationship to system dependability [DT89], and fault-injection [ACC+93] also is essential to understand the context of our work. Our program-coverage metrics are most similar to those used in dataflow testing [RW85]. These references have been discussed in Section 2.1.

Our fault-injection experiments most closely resemble those measuring responses to errors using traditional program-coverage metrics. Tsai et. al [THZ+99] placed breakpoints at key program points along known execution paths and injected faults at each point, (e.g., by corrupting a value in a register). Their work differs from ours in its goal, the kinds of faults injected, and their definition of coverage. The primary goal of their approach was to increase fault activations and fault coverage, not to increase program coverage. They injected a set of hardware-centric faults such as corrupting registers and memory; these faults primarily affected program state, not communication with the operating system or I/O hardware. They used a basic-block definition of program coverage, rather than measuring coverage of a program-level construct such as a `catch` block. Bieman et. al [BDL96] explored an alternative approach where a fault is injected by violating a set of pre- or post-conditions in the code, which are required to be expressed explicitly in the program by the programmer. This approach used branch coverage, a program-coverage metric.

In the terminology of Hamlet's summary paper reconciling traditional program-coverage metrics and probabilistic fault analysis [Ham94], our work can be classified as a probabilistic input sequence generator, exploring the low-frequency inputs to a program. Using the terminology presented by Tang and Hecht [TH97], which surveyed the entire software dependability process, our method can be classified as a stress-test, because it generates unlikely inputs to the program.

6 Conclusions

We have posed what we believe to be a new challenge in the field of techniques for development of highly available systems: to determine whether all of the fault-recovery code in a Web services application has been exercised on an appropriate set of faults. We have presented our fault-catch coverage metric, which formalizes what it means to meet this challenge successfully, and have shown that it is possible to instrument programs to collect coverage information at run-time. Our metric combines ideas of testing software in response to injected faults, developed by the dependability community, with ideas of testing for coverage of specific program constructs, developed by the software engineering community.

We also have developed an API that allows the program being tested to direct a fault-injection engine and have extended Mendosusto respond to this API. We have described compiler analyses that can be applied to Java source or bytecodes in order to instrument codes to direct fault injection to produce high fault-catch coverage.

Our preliminary case study results with Muffin indicate that our approach is highly effective for faults that do not create latent errors (i.e., 100% coverage), and somewhat effective for faults that do (i.e., covering 4 of the 7 **NIC_DOWN**/`catch` combinations). Next we plan to enhance our approach to achieve better coverage in the presence of latent errors and to study issues of testing of distributed applications.

References

[ABB+02] Patterson D. A., A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, University of California, Berkeley, March 2002.

[ACC+93] Jean Arlat, Alain Costes, Yves Crouzet, Jean-Claude Laprie, and David Powell. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Transactions on Computers*, 42(8):913–923, August 1993.

[AG97] K. Arnold and J. Gosling. *The Java Programming Language, Second Edition*. Addison-Wesley, 1997.

[AS00] M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. Technical Report RC 21789, IBM T.J. Watson Research Center, July 2000.

[BCS69] W. G. Bouricius, W. C. Carter, and P. Schneider. Reliability modeling techniques for self repairing computer systems. In *In Proceedings of the 24th National Conference of the ACM*, pages 295–309, March 1969.

[BDL96] J. Bieman, D. Dreilinger, and L. Lin. Using fault injection to increase software test coverage. In *Proc. 7th Int. Symp. on Software Reliability Engineering (ISSRE'96)*, pages 166–74. IEEE Computer Society Press, 1996.

[Bin99] Robert V. Binder. *Testing Object-oriented Systems*. Addison Wesley, 1999.

[BS96] D. Bacon and P. Sweeney. Fast static analysis of c^{++} virtual functions calls. In *Proceedings of ACM SIGPLAN Conference on Object-oriented Programing Systems, Languages and Applications (OOPSLA'96)*, pages 324–341, October 1996.

[CCH+99] Michel Cukier, Ramesh Chandra, David Henke, Jessica Pistole, and William H. Sanders. Fault injection based on a partial view of the global state of a distributed system. In *Symposium on Reliable Distributed Systems*, pages 168–177, 1999.

[DGC95] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy. In *Proceedings of 9th European Conference on Object-oriented Programming (ECOOP'95)*, pages 77–101, 1995.

[DJM96] Scott Dawson, Farnam Jahanian, and Todd Mitton. ORCHESTRA: A fault injection environment for distributed systems. Technical Report CSE-TR-318-96, 26 1996.

[DT89] Joanne Bechta Dugan and Kishor S. Trivedi. Coverage modeling for dependability analysis of fault-tolerant systems. *IEEE Transactions on Computers*, 38(6):775–787, June 1989.

[ecl] The Eclipse IDE. See `http://www.eclipse.org/`.

[Ele99] Electronic News. Ebay Outages Cast Clouds on Sun. http://www.electronicnews.com/enews/Issue/1999/06211999/25ebayah.asp, June 1999.

[GC01] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6), 2001.

[Gra] Jim Gray. Dependability in the Internet Era. Keynote presentation at the Second HDCC Workshop.

[Ham94] Dick Hamlet. Foundations of software testing: dependability theory. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of software engineering*, pages 128–139. ACM Press, 1994.

[HC94] Herbert Hecht and Patrick Crane. Rare conditions and their effect on software failures. In *In Proceedings of the Annual Reliability and Maintainability Symposium*, pages 334–337, Anaheim, CA, January 1994.

[HSR95] S. Han, K. Shin, and H. Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems, 1995.

[KKA92] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A tool for the validation of system dependability properties. In *FTCS-22: 22nd International Symposium on Fault Tolerant Computing*, pages 336–344, Boston, Massachusetts, 1992. IEEE Computer Society Press.

[KKI99] M. Kalyanakrishnam, Zbigniew Kalbarczyk, and Ravishanka Iyer. Failure Data Analysis of a LAN of Windows NT Based Computers. In *Proceedings of the 18th Symposium on Reliable and Distributed Systems (SRDS '99)*, 1999.

[LMN+02] Xiaoyan Li, Richard P. Martin, Kiran Nagaraja, Thu D. Nguyen, and Bin Zhang. Mendosus: A SAN-Based Fault-Injection Test-Bed for the Construction of Highly Available Network Services. In *Proceedings of the 1st Workshop on Novel Uses of System Area Networks (SAN-1)*, Cambridge, MA, January 2002.

[LPH01] D. Liang, M. Pennings, and M.J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for java. In *Proceedings of the 2001 ACM SIGPLAN - SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79, June 2001.

[Mar95] Brian Marick. *The Craft of Software Testing, Subsystem Testing Including Object-based and Object-oriented Testing*. Prentice-Hall, 1995.

[MRR02] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analysis. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 1–11, 2002.

[muf] The Muffin world wide web filtering system. See `http://muffin.doit.org/`.

[Mye79] Glenford J. Myers. *The Art of Software Testing*. John Wiley and Sons, 1979.

[Reu99] Reuters. Britannica Competitors Study Net Strategy. http://news.cnet.com/news/0-1005-200-1435449.html, November 1999.

[RMR01] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java using annotated constraints. In *Proceedings of the Conference on Object-oriented Programming, Languages, Systems and Applications*, pages 43–55, 2001.

[RW85] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[Set96] Ravi Sethi. *Programming Languages, Concepts and Constructs, 2nd Edition*. Addison Wesley, 1996.

[Sil00] SiliconValley.internet.com. Ebay Outage Twice This Week. http://siliconvalley.internet.com/news/article/0,,3531_435741,00.html, August 2000.

[SM] Sun-Microsystems. Java development kit 1.2. See `http://java.sun.com/products/jdk/1.2/docs/api/`.

[SVS+88] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson, and T. Lin. FIAT w fault injection based automated testing environment. In *Proc. 18th Int. Symp. on Fault-Tolerant Computing (FTCS-18)*, pages 102–107, Tokyo, Japan, 1988. IEEE Computer Society Press.

[Swe00] T. Sweeney. No time for downtime – it managers feel the heat to prevent outages that can cost millions of dollars. *InternetWeek*, (807), April 2000.

[TH97] Dong Tang and Herbert Hecht. An approach to measuring and assessing dependability for critical software systems. In *In Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pages 192–202, Albuquerque, NM, November 1997.

[THZ+99] T. Tsai, M. Hsueh, H. Zhao, Z. Kalbarczyk, and R. Iyer. Stress-based and path-based fault injection. *IEEE Transactions on Computers*, 48(11):1183–1201, November 1999.

[TP99] Nisha Talagala and David Patterson. An Analysis of Error Behaviour in a Large Storage System. In *Proceedings of the Annual IEEE Workshop on Fault Tolerance in Parallel and Distributed Systems*, April 1999.

[TP00] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the Conference on Object-oriented Programming, Languages, Systems and Applications*, pages 281–293, October 2000.