What's Wrong With Fault Injection As A Benchmarking Tool?

Philip Koopman
ECE Department & ICES
Carnegie Mellon University
Pittsburgh, PA, USA
koopman@cmu.edu

Abstract

This paper attempts to solidify the technical issues involved in the long-standing debate about the representativeness of fault injection as a tool for measuring the dependability of general-purpose software systems. While direct fault injection seems appropriate for evaluating fault tolerant computers, most current software systems are not designed in a way that makes injection of faults directly into a module under test relevant for dependability benchmarking. Approaches that seem more likely to create representative faults are ones that induce exceptional operating conditions external to a module under test in terms of exceptional system state, exceptional parameters/return values at an API, failed system components, or exceptional human interface inputs.

1. Introduction

Fault injection has long been used as a way to evaluate the robustness of computer systems. Traditional fault injection techniques involve perturbing or corrupting a computer's memory or operating elements to create a fault. Mechanisms to inject faults include heavy-ion radiation, probes that inject electrical signals into hardware circuits, mutation of programs, and software probes that corrupt system state as programs are being executed. These techniques are widely considered useful in evaluating a system's response to a hardware fault such as a radiation-induced bit value inversion or a "stuck-at" fault. They also can be effective at exercising various fault handling mechanisms, both software and hardware.

While previous generation system dependability was often dominated by hardware faults, in the vast majority of modern computer systems it is widely believed that software problems are a more frequent cause of system-level failures. Thus, attempts to measure and improve overall system dependability are in need of a way to measure software dependability. Over time, fault injection has come to be used as a controversial technique for assessing software fault tolerance and system-level dependability.

It is arguably the case that the question of representativeness is the single biggest open issue in fault injection today. Fault injection, especially software-based fault injection, is a reasonably convenient and effective way to evaluate systems. However, it is not always clear what fault injection results mean for fielded systems. Proposing the use of fault injection to benchmark system dependability typically triggers vigorous debate, often with polarized viewpoints.

This paper seeks to identify a possible source of the large gap between opposing views in the debate on fault injection representativeness as well as place into perspective some of the strengths and weaknesses of fault injection approaches. While this paper presents opinions rather than scientific facts, it is based on experiences from extensive interactions with industry and the technical community as part of the Ballista robustness testing project as well as two years as co-chair of the IFIP 10.4 WG SIG on Dependability Benchmarking.

This paper has been written to start a discussion rather than end one. None of the issues discussed below are really quite as black and white as they are portrayed to be. In particular, any tool that affords a different view of a system for analysis brings with it some benefits. However, the disparity between benefits claimed by some fault injection proponents versus the lack of value perceived by some fault injection opponents suggests that a clear and precise articulation of issues would help focus the debate. It is our hope that this paper will put future discussions on this topic on a more solid technical foundation.

2. Fault injection as a fault tolerant computing evaluation tool

One way to understand fault injection is to hypothesize a system that would be sure to detect and recover from all possible fault injection experiments (*i.e.*, one that would be perfect if evaluated using fault injection, although of course in practice such perfection is not attainable). We assume the classical situation in which faults are only injected within the boundaries of a software module under test (other situations are discussed Section 5). Thus, a software fault injection experiment consists of injecting a fault into the data or program of a module and evaluating whether the

result matches that of a so-called golden run of an uncorrupted copy of that same module. Compared to traditional software testing techniques, this method of evaluation has the distinct virtues of being relatively inexpensive to develop and execute, as well as requiring minimal information about the software being tested.

A software module which would be completely invulnerable to such injected faults would have to incorporate exception handlers and acceptance tests. Exception handlers would catch and convert any possible exceptions generated by an injected fault into a clean error return operation followed by a subsequent recovery operation. Because arbitrary changes can be made to a module under test by fault injection, exceptions generated can potentially include those from illegal instructions, memory protection violations, and other activations of machine checks. However, not all program state corruptions result in exceptions. Thus, acceptance tests would also be required in order to catch and recover from any possible incorrect (but non-exceptional) result due to program or data mutation.

The above description of an idealized system should help explain why fault injection is historically popular in the fault tolerant computing community. A system that can withstand an arbitrary fault can be said to be dependable without concern for the specifics of the situation it is being used in. Systems that employ fault tolerant techniques often have hardware support that tends to detect or correct injected faults, as well as software that is designed for recoverability after failure and for self-checking of computational results. Note that the requirement for an acceptance test that can detect an arbitrary fault can be difficult to meet, and often leads to multi-version software design and other expensive implementation techniques that can be justified only for the most critical systems.

Perhaps surprisingly, the widely used technique of input value validity checking is not sufficient to assured dependability against arbitrary fault injections. In the common case, an injected fault can be expected to create an exception or incorrect value after a module performs validity checking on input values. Thus, software would have to be able to handle arbitrary corruptions of data values even if they had previously passed input validity tests. Adding input value tests can help strengthen a weak system by adding indirect acceptance tests so that one module's output gets tested as it becomes the next module's input. But, it does not seem that a fully robust system can rely upon them exclusively.

Put another way, all exceptions that can be generated by hardware in any circumstance must be handled, even ones that appear to be impossible based on the software design. For example, a null pointer check might be changed by an injected fault into an instruction that corrupts a pointer instead of ensuring it is valid – there is no assurance that input validity checks will suffice.

The issue of what happens when exception handlers or acceptance tests are themselves corrupted is of course relevant. One can argue that the chance of a value being changed after its validity check is larger than the chance that an arbitrary fault will result in both a corrupted value and the propagation that value past an acceptance test. But regardless of that argument, it is clear that validity checks alone do not suffice to ensure dependability for an arbitrary fault model.

Although it is still an open question whether direct injection of arbitrary faults leads to representative results with regard to software dependability evaluation, systems built to withstand arbitrary fault injections successfully can clearly endure very difficult operating conditions. And in fact these are the kind of systems that the fault tolerant computing community has long dealt with. While it is impossible to be perfectly robust under every conceivable set of concurrently injected faults, systems that perform well during a fault injection campaign truly deserve to be called robust, but are likely to achieve this result at significant expense.

The controversial question of concern is: should we measure all systems by their response to arbitrarily injected faults? Or, for that matter, is it appropriate to compare any systems by such a standard?

3. Everyday, but mission-critical, software

In most computing, even many mission-critical computing applications, the expense of creating and operating a completely fault tolerant computing system can be too high. Thus, optimizations are employed to reduce development and run-time costs. While programs employing these optimizations can at times be less dependable than more traditional fault-tolerant software systems, there is no essential reason why this need be the case for the types of faults actually encountered in everyday operation.

First and foremost, general purpose computing assumes that hardware faults are rare, are detected by hardware mechanisms, and need not be dealt with by application software. This is by and large a reasonable set of assumptions. In current systems, system crashes and problems created by software failures seem to outnumber hardware failures by a large margin. Furthermore, well known and affordable hardware techniques such as using error detection coding on memory and buses are available to reduce the chance of undetected hardware faults and, if desired, provide automatic recovery from likely hardware faults. (Whether customers decide to actually pay for hardware reliability features is in fact an issue, but arguably one that won't be resolved by creating even more hardware or software reli-

ability mechanisms that have non-zero cost to implement or execute.) Mainframe computers have provided very reliable hardware for many years – thus the capability to provide effectively failure-free hardware environments is more a matter of economy than practicability.

Given that application software can consider hardware to be failure-free for practical purposes, significant and important optimizations can be employed to simplify software and speed up execution. The general idea behind these optimizations is to avoid repeating validity checks and avoid providing unnecessary exception handlers.

Given failure-free hardware, validity checks need only be computed once for any particular value. As a simple example, if a memory pointer must be validated before it is dereferenced, it suffices to check pointer validity only once upon entry to a module (and after pointer updates) rather than every time that pointer is dereferenced. Many such checks can be optimized using the general technique of common subexpression elimination. (Compilers might well perform this optimization even if checks are explicitly included before each pointer use in source code.) Furthermore, if a set of related modules is designed to be used together, such checks can be optimized across module boundaries and only invoked at external entry points into the set of modules. Similarly, if exception handlers are preferred to data validity checks, these need only be provided at external software interfaces.

Of course it is well known that data validity checks used in practice are typically imperfect and could easily be improved in many systems. However, the important point for present purposes is that this is the preferred approach to attaining robustness in such systems, and that most such systems typically have at least some validity checks of this nature in place. For example, in mission-critical software systems, it is common practice to add additional "wrappers" to perform validity checks at strategic places in the system, such as before every call to the operating system.

An additional situation that can be considered an optimization is that acceptance tests used at the end of module execution are often omitted or reduced to rudimentary form. Instead, what is supposed to happen is that correctness checks are moved from run time to development time, and take the form of software testing suites and compiler analysis to ensure software conformance to specifications. Given the assumption of no hardware failures, this is a reasonable way to reduce fielded software size and execution time while ensuring correct operation. It is of course recognized that test suites are generally imperfect; however this does not change the fact that this is the prevalent approach in ordinary software systems. Even in systems with rigorously defined exception-handling interfaces, it is commonly the case that only exceptions that are expected to be generated are supported, documented, and tested. Exceptions that are impossible to generate, based on analysis of source code and a failure-free hardware assumption, are usually not handled gracefully since doing so would be seen as a waste of development resources within a finite budget of time and money.

Now let us say that a software module developed according to the above implementation philosophy is subjected to fault injection as a way to exercise the system in an attempt to find software dependability problems. The important question for this case is how to interpret a failure that is created by a fault injected into the code or data spaces of a module under test. There are two main cases of interest to consider based on the fault injection outcome.

(1) The injected fault produces a valid, but incorrect, intermediate result that propagates to the output. If there are no correctness defects in the module under test, this can only correspond to incorrect data being presented at the external inputs of the module. That situation does not correspond to a defect of any kind in the module under test because the program that was actually written does not produce the same output as the fault-injected output for that particular system state and set of input conditions, and is therefore not representative of expected failures. Saying that an ordinary software module is defective because it gives a different answer when processing internally corrupted data (or executing arbitrarily modified instructions) than when processing uncorrupted data is, in the general case, unreasonable.

A different argument that is sometimes made is that such a situation represents a *hypothesized* fault (similar in intent to the concept of "bebugging" via source code mutation in the software testing community). In this case such a fault can only be argued to be useful in evaluating the effectiveness of the test suite. But even then interpretation of results must be made carefully if tests have been designed in a "white box" manner, taking into account the structure of the module being tested. It might not be reasonable to criticize a test suite for failing to find a defect in a program that has been mutated to have a different structure or functionality than the one the tests were designed for. In other words, the concept of optimization can and does extend to designing tests that are only relevant for the software that has been specified in light of reasonable and common defects. Furthermore, this use of fault injection requires availability of a test suite, which is seldom the situation encountered when fault injection is used as a dependability metric.

(2) The injected faults produce an inappropriate exception or crash. In this case determining the correctness of module functionality is simplified by applying a default specification that the module should not produce an undocumented (or unrecoverable) exception and should not crash or hang. These sorts of module failures can be legitimately called robustness failures, which reduce software depend-

ability, but only if they could plausibly happen during the execution of a real application program.

Consider a pointer validity check to illustrate the issue of representativeness in this situation. A program might check a pointer passed as an input parameter at the beginning of a module, and then dereference it many times during execution without altering its value. Given that hardware faults are not under consideration, that program can never suffer a memory protection violation by dereferencing that pointer. However, a fault injection campaign might well create an invalid pointer and a subsequent exception or crash. It is difficult to claim that this result measures software dependability, because it could never happen during production use. Even more importantly, it is unreasonable to claim that it is representative of a hypothesized software robustness defect, because in fact this is a case in which the software was specifically designed to be robust to the very type of fault that was injected!

To further illustrate the problem with representativeness in this case, consider a situation in which a null pointer check has been omitted from a piece of off-the-shelf software available only in executable form. The traditional solution to cure robustness failures of this type would be to add an external wrapper to the module that checks for pointer validity. Adding that wrapper makes the software robust to null pointer inputs. But, this wrapped module would still suffer an exception in response to a fault injection that creates a null pointer after the wrapper performs its check. Thus not only would robust software appear to have robustness failures under a fault injection experiment, but hardened software would similarly appear to be non-robust.

Software that performs acceptance tests might tend to appear more robust under fault injection than software that just performs input validity checks. This is because any injected faults that are activated during the execution of a module would, theoretically, be detected and handled by an acceptance test. However, optimizations would still introduce vulnerabilities to fault injection, such as not implementing exception handlers for exceptions that are impossible in fault-free hardware operation. For an invalid pointer example, there is no reasonable basis for incurring the cost of memory address exception handlers if all pointers are known to be valid based on previous validity checks.

Thus, direct fault injection experiments are not suitable for general-purpose software robustness benchmarking. They are unlikely to show an improvement in robustness when proven techniques are used to improve software robustness via filtering out invalid or exceptional input values. While it could then be claimed that fault injection would alternately test whether all possible exceptions were caught by a module, this approach has a similar problem in that fault injection is likely to generate exceptions that can't possibly happen in real program runs. Furthermore, even adding exception handlers will not necessarily catch injected faults that generate unexpected exceptions or, worse, disable or subvert the exception handler itself.

4. The importance of realistic fault activation

The key problem with using direct fault injection on a software module is that doing so does not take into account whether the injected fault can be activated by inputs during real execution, and similarly whether it can slip by any available exception handler. Mainstream software development, even for critical systems, traditionally achieves robust operation in large part by restricting the ability of exceptional values to activate potential fault sites via a set of input validity checks. It then seeks to handle only those exceptions that can actually be encountered (*i.e.*, ones for which input checks are not implemented or cannot provide complete coverage) to minimize software complexity.

Of course most software is certainly not perfect. Not all values are checked before being used. Not all exceptions that can be generated are handled gracefully. However, it appears that fault injection has trouble distinguishing whether a general purpose software module (as opposed to a specially created fault tolerant software module) would in fact be operationally robust.

To use an analogy to the problem under consideration, consider the task of making a camping tent absolutely dark, with no light entering whatsoever, perhaps for the purpose of photographic film processing. A single layer of black plastic sheeting might suffice for this task, but might also be imperfect or develop holes while in use. A fault tolerant computing analogy might well involve using multiple layers of plastic sheeting to reduce the probability of aligned multi-layer holes. Fault injection would then involve putting a few small holes in individual layers to ensure that light still did not penetrate, simulating the process of holes appearing due to wear and tear during use.

By the same analogy, a fault injection technique applied to a cost-sensitive situation would be to prick holes in a single layer plastic tent and assert that light indeed penetrated into the tent when a hole was made (assume that after each such hole is evaluated it is automatically patched). Furthermore, fault injection might prick and illuminate holes in portions of the tent which were not exposed to light in the first place such as plastic buried under dirt at the edges of the tent. (A more extreme version of fault injection would be to shine a flashlight inside the tent and assert that light had entered; but an expected response in that case would be to eject the person holding the flashlight!)

While pricking holes in a single-layered plastic tent indeed serves to illustrate the vulnerability of using a single layer of plastic, one should not be surprised that owners of such a tent don't see much value in a fault injection exer-

cise. A more effective approach for their situation would be to create ways to identify existing holes so they could be patched, perhaps by entering the tent and patching places where light leaked in, analogous to installing software wrappers to improve dependability. Thus, the core problems of a fault-injection based approach in this analogy of an *optimized, cost-sensitive system* are (1) identifying defects that aren't in the system *as it is actually implemented*, and (2) creating false alarms to such a degree that effort is arguably better spent on other activities. Saying that optimizing systems for cost can lead to less robust systems is of course a valid point, but does not change the fact that most purchasers demand optimized systems; insisting that no such systems be implemented is simply unrealistic.

The key problem is one of whether activation of a given potential fault could actually occur in a real system under real operating conditions, or if it can occur, whether it takes place under a condition that the designers of the system feel justified in ignoring. Direct fault injection creates situations that might be impossible to achieve in real execution, by for example creating an exceptional value just after a validity check. Without a measure of whether activation of an injected fault is possible in a real system, it is difficult to use the results of direct fault injection for everyday software. For this reason, direct fault injection is ultimately no substitute for a good development and test process. It can, however, help in designing a module to tolerate design faults contained in *other* modules and its execution environment as discussed in the next section.

5. Representative uses of fault injection

Given the above discussion, it would be a mistake to conclude fault injection is of no use in evaluating the dependability of mainstream software. The real issue is using fault injection in a manner appropriate to the relevant assumptions and development model for any particular system under consideration. For systems designed to tolerate generalized failures as described in Section 2, fault injection seems like a useful tool to use in overall system evaluation. But lack of tolerance for directly injected faults does not demonstrate software undependability; it merely fails to make a case for software being dependable in the face of arbitrary runtime faults.

Some of the ways in which fault injection can be helpful are listed below. Note that these techniques avoid injecting faults directly into the module under test, but rather inject faults into external interfaces or the surrounding environment. Thus all such approaches have the virtue of being non-invasive with respect to the actual execution of the module under test and thus present a fault model that does not assume a particular approach is used within a module for providing dependable operation.

- **Callee interface fault injection.** Faults can be injected into the inputs of a module to check for out-of-specification responses. The presumption here is that some external software defect exists that feeds an exceptional value (one that is invalid rather than merely incorrect) to the module under test. This approach is only representative to the degree that injected faults could really happen in operation. It tends to be most useful for very widely used application programming interfaces (APIs) for which all uses cannot possibly be foreseen, as well as for faults injected that are common results of programming errors (*e.g.*, submitting a null value as a pointer parameter to a procedure call). A variation on this theme is testing for security vulnerabilities such as buffer overflow vulnerabilities.
- **Caller interface fault injection.** Faults can be injected in values returned to the module under test from other software that it calls via external calling interfaces. Representativeness in this situation can be tricky to determine if the module is only destined to run on one version of supporting software if that software cannot actually generate the faults used for testing. However, the utility of this approach increases if the module under test must be portable or the underlying software can be expected to be upgraded or revised over time.
- **Forced resource exhaustion and environmental exceptions.** Exceptional conditions such as having no allocable memory or a full hard disk can stress a software module under test in terms of its ability to process exceptions. This can be seen as a special case of fault injection into the external caller interface as just described, but is in general an important case to cover.
- **Failure of cooperating processes**, tasks, and computers in a distributed system.
- **Configuration, maintenance**, and other failures in the supporting infrastructure and execution environment, including failed hardware components such as disk drives, missing or incorrectly configured software supporting the execution of a module under test, and even installation of incompatible versions of software.
- **User interface failures**, in which users create exceptional inputs or operational situations.

The above approaches are all relevant to some degree to both fault tolerant software systems as well as general-purpose, optimized software systems because they deal with factors outside the control of designers performing software module optimization.

6. Conclusions

Direct injection of faults into a software module can provide evidence of software dependability. Certainly any piece of software that can tolerate a large fraction of possi-

ble arbitrary changes to its operation and still provide correct answers is dependable, and is likely to provide extensive exception handling capabilities to provide that dependability. However, this does not make such fault injection suitable as a benchmarking technique in general, because much software is not designed to withstand fault injection, but rather is optimized assuming correct hardware operation.

Using direct fault injection as a dependability benchmark commits the logical fallacy of "denying the antecedent" (this fallacy is an argument of the form: "If A then B; Not A, thus not B"). In this instance the specific logical fallacy would be: "if software performs well under fault injection then it is dependable; a particular piece of software performs poorly on a fault injection test, thus it is not dependable." Because much software is optimized in ways that preserve dependability for the expected case of failure-free hardware, one *cannot* conclude that software that does poorly at direct fault injection is undependable. The most that can be inferred is that the results provide suggestive evidence that software does not use exception handling as its primary means of providing dependability (but even that statement assumes that injected faults did not affect exception handling itself).

It is possible that the controversy over fault injection continues because of disparate world-views of how robust software should be created. For systems that must achieve comprehensive fault tolerant operation, including withstanding hardware faults, injecting faults directly into a module under test can yield insight into operation in the face of equipment faults. Traditional fault tolerant software, and in particular software that has comprehensive acceptance test support and exception handling, might be evaluated using fault injection techniques. The ability to demonstrate how such injected faults correspond to measurements of software defects rather than hardware defects is still a research topic. But, the arbitrary nature of injected faults that are permitted to activate at the software level rather than being compensated for by hardware mechanisms can be argued to provide a substantial exercise of the software fault tolerance designed into a system.

General-purpose software tends to be optimized to eliminate redundant validity checks and superfluous exception handlers under the assumption that hardware faults are both rare and detected without the involvement of application software. This means that a fault that is injected might well result in a failure that is not representative of the particular software module under test (*i.e.*, the injected fault might create an exceptional value at a point in a computation where such a value would have already been caught by validity checks). Direct fault injection experiments on general-purpose software tend to underestimate the degree of operational robustness available, and in particular are poorly suited to measuring the improvement of robustness afforded by adding input value checks to software interfaces via code modification or addition of robustness wrappers. Thus, fault injection results in this case distinguish *which approach* was used to ensure software dependability (validity checks versus exception handling) rather than solely *whether* the software is in fact dependable.

For systems in which hardware is presumed to have enough built-in capabilities to offer fault-free operation for practical purposes, direct fault injection seems to offer little evaluative benefit in the absence of an accurate fault activation analysis. But, fault injection into the surrounding environment seems attractive as a way to characterize exceptional condition response. It is important in every such experiment to clearly state several things for the results to be meaningful: the fault hypothesis being evaluated, the correspondence between the faults being injected versus that fault hypothesis, and the assumptions being made about the structure of software being evaluated. And finally, it is important to avoid confusing the ability of fault injection to demonstrate robustness with the fact that lack of such a demonstration does not prove that software is non-robust within reasonable operating environments.

7. Acknowledgments

The reader will note a lack of scholarly references in this paper. The important parts of the relevant discussions have taken place in meetings and e-mails over a period of years, and thus are impractical to cite or even attribute. Citing only scholarly publications as sources of various current and historical ideas would overlook that many of these ideas circulated long before they were published or aren't formally published, and would likely provoke a variety of divisive responses from authors who object to our interpretation of their work. Thus we simply assume that the reader is familiar with fault injection as a general technique, and we acknowledge that none of the technical ideas in this paper are new; we simply summarize experiences and possibly present them in a new light.

Discussions with the members of the IFIP WG 10.4 SIG on Dependability Benchmarking and in particular with Henrique Madeira have been invaluable in forming the thoughts presented in this paper. This paper is, however, solely the opinion of the author. This material is based upon work supported in part by DARPA and US Army Research Office under Award No. C-DAAD19 01-1-0646, and the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the view of the sponsoring agencies.