Why do Some (weird) People Inject Faults ?

João Carreira, João Gabriel Silva
Dependable Systems Group, Dept.of Computer Engineering
University of Coimbra, Portugal
{jcar,jgabriel}@dei.uc.pt

There are some research corners in computer engineering whose usefulness is sometimes hard to understand even for experienced engineers. Fault Injection is one of those areas. Since the day we built our very first Fault Injection tool we hear the same jokes: *"Why the heck are you injecting faults ? don't you think there are already enough ?"*. Fortunately work in fault injection has progressed and nowadays this is a technique used by many of the biggest computer manufacturers. We'll try to show briefly why is Fault Injection important and what's its practical use.

It is common nowadays to hear about computer systems faults in the media. The reasons can be very diverse: from the blackout of America Online (AOL) affecting 6-million users to a software fault in Ariane 5 that cost $500 billions to the European Space Agency (ESA). But what exactly are we talking about when we talk about faults ? Simplifying things a little bit, a fault is a phenomenon that can cause a deviation in an hardware or a software component from its intended functions [2].

Hardware faults that occur during system operation are categorized mainly by their duration. *Permanent* faults are caused by irreversible component damages due to exhaustion, improper manufacturing or misusage. These faults can only be recovered by replacing or repairing the faulty component. A simple example is a chip that burns in your network card causing it to stop working. *Transient* faults, on the other hand, are triggered by environmental conditions such as voltage fluctuation, electromagnetic interferences or radiation. These faults usually do not cause any lasting damage in the affected component, although they can cause the system to change to an erroneous state. According to several studies, transient faults occur much more frequently than permanent ones and they are also much more difficult to detect. In fact these faults are the real headache of engineers working on fault tolerance. The last class of *Hardware* faults are *Intermittent* faults. They occur due to unstable hardware or varying hardware states and can be repaired by replacement or redesign.

Finally, *Software* faults are caused by incorrect specification, design or coding of a program. Every software engineer knows that a software product is bug-free just until the next bug is found. Many of these faults can be latent in the code and show up only during operation, specially under heavy or unusual workloads and timing contexts or due to a phenomenon known as *process aging* [3]. The bugs of non-deterministic nature which are also called *Heisenbugs* (as opposite to *Bohrbugs* that predictably lead to failures) are the most difficult ones to eliminate by verification, validation or testing and therefore they are something the system has to live with. Curiously, most of the computer system faults are attributed either to *Software* faults or *Permanent* faults. This doesn't mean that *Transient* and *Intermittent* faults occur less frequently than software ones but simply that they are much more difficult to track.

So, faults can have diverse origins, but after all what's the role of Fault Injection in this story ? The keywords are **Dependability Validation** and **Software Testing**. Fault injection is mostly used to validate fault tolerance mechanisms in computer systems. Imagine a computer system for the space shuttle, railway control, nuclear power plants, fly-by-wire, or medical life-keeping. What these systems have in common is that they require high-levels of dependability. The consequences of a fault can be disastrous in terms of human lives or economic losses. It is clear that such dependability requirements cannot be met solely by careful design, quality assurance, shielding or other *Fault Avoidance* techniques. The assumption that faults could be completely avoided is unrealistic and therefore the computer system must be able to provide the expected service in the face of faults - this is the purpose of *Fault Tolerance* (FT).

Fault Tolerance can range from very complex to reasonably simple mechanisms. For example, the space shuttle uses a Module Redundancy a scheme where computer systems designed by different unrelated teams from the same specification run in parallel using the same input data. A device known as a *voter* monitors the output of the three systems looking for any discrepancy. If one is detected the result which got the majority is used while the deviating result is considered as anomalous. An example of a simple fault tolerance mechanism is a *watchdog* process [5] that monitors applications for execution problems and performs automatic restarts in case of application *crashes* or *hungs*.

Whatever the complexity of the fault tolerance mechanisms, its first requirement is to have the ability to detect faults, or more precisely the *errors* caused by the faults. Parity checking is a simple example of an error detection mechanism. Once an error has been detected, the affected component has to be identified (diagnostic) and possibly isolated through system reconfiguration and/or other recovery actions preferably in an automatic and totally transparent way- not an easy task.

When the fault tolerant mechanisms are in place and their different modules tested the problem of overall system testing and validation shows up. How to efficiently test and validate the fault tolerant system as a whole ? This is important both for the manufacturers and clients of fault tolerant systems. Manufacturers want to be able to advertise the level of fault tolerance of their systems and clients want critical equipment to be certified. But how to certify the level of dependability of such systems? Certification agencies such as the German TUV are currently facing this problem and trying to devise the most appropriate methodology to do so. The use of analytical modeling is very difficult as the mechanisms involved in the fault activation and propagation process are highly complex and are not completely understood in most of the cases. Furthermore, the simplifying assumptions usually made to make the analysis tractable reduces the usability of the results achieved by this method.

One practical and efficient way of performing this kind of validation is experimentally, you guessed it, using *Fault Injection* [1]. The principle is to insert faults in the system as close as possible to the faults

that can occur in the field and check how the system reacts. This is useful in one hand to test if the fault tolerance mechanisms are behaving as specified (validation) and on the other hand to assess the coverage of the mechanisms, i.e. what percentage and what kind of faults they can handle (evaluation). It is relatively easy to induce *Permanent* faults and check how the system behaves; it's just a matter of disabling a component, e.g. disconnecting the network card. With *Transient*, *Intermittent* and *Software* faults it is not that easy. The real problem is in devising the kind of faults that should be injected and whether they correspond to faults that can really occur. The hard approach to *Fault Injection* is to disturb the hardware directly. This can be done by momentaneously flipping bits at the chip pins, varying the power supply or even bombing the system/chips with heavy ions. These methods are believed to cause real hardware *Transient* faults but unfortunately don't give much help for *Software* faults. Furthermore, the high complexity and the very high speed of the processors available today makes the design of the special hardware required by the above approach very difficult, or even impossible. Another approach is to build a simulator of the system and inject faults by bit flipping directly in the simulation model. This allows a fine control over the timing, the type of fault, and the affected component in the system. However, it involves an enormous development effort to build the simulator which is not compatible with time-to-market requirements of most companies.

A recent approach that is being increasingly used as an alternative to the others is to insert the faults in the system using a software tool - a Software Implemented Fault Injector or SWIFI for short [4]. This tool is basically used to interrupt the execution of the critical system software and flip bits in different parts of the system such as the processor registers. the memory, or the application code. The advantages of this approach is its low complexity, low development effort, low cost (no specific hardware is needed) and increased portability. An additional advantage is that built-in mechanisms for debugging and performance monitoring that are current in most modern microprocessors can be used to emulate hardware faults with a great precision. A SWIFI tool is usually a very bizarre piece of software which makes use of all possible processor and system hooks to create an incorrect behavior in a controlled manner.

Another thing that makes SWIFI different from the other techniques is its ability to emulate software (application level) faults, and this takes us to another area - **Software Testing**. Mission critical applications require extensive testing. specially to catch the so-called *Heisenbugs* which occur during unusual and limit situations. There's nothing like executing those applications under the adverse conditions created by a fault injector to force software faults to show-up [6].

After all Fault Injection seems to have some important applications. In fact, a fault injector is a tool of invaluable help for dependability engineers. The route can look tortuous, but effectively we can say that some people artificially inject faults in order to understand and tolerate real faults. Confused ?

[1] Jean Arlat et al., "Fault Injection for Dependability Validation: A Methodology and Some Applications", IEEE Trans. on Software Engineering., Vol 16, No 2, Feb. 1990, pp. 166-182.

[2] J.C.Laprie (editor), "Dependability: Basic Concepts and Terminology", vol.5 of Dependable Computing and fault Tolerance, Springer-Verlag, 1992.

[3] Yennun Huang, C.Kintala, Nick Kolettis and N.Fulton, AT&T Bell Laboratories, "Software Rejuvenation: Analysis, Module and Applications", 25th Fault Tolerant Computing Symposium, 1995.

[4] João Carreira, Henrique Madeira, João Gabriel Silva. "Xception: Software Fault Injection and Monitoring in Processor Functional Units" Proc.of DCCA'95, Working Conference on Dependable Computing for Critical Applications, Urbana-Champaign, USA, September 27-29 ,1995.

[5] João Carreira, Dino Costa, João Gabriel Silva, "Fault Tolerance for Windows Applications", BYTE Magazine, Core Technology - Operating Systems column, February 1997.

[6] Jeffrey Voas, "Software Fault-injection: Growing 'Safer' Systems", In Proc. of IEEE Aerospace Conference, Snowmass, February, 1997.