Belegarbeit

Implementation of a Fault Injection Framework for Fiasco.OC

Martin Unzner

14th January 2013

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Björn Döbel

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 14. Januar 2013

Martin Unzner

Contents

List of Figures

List of Tables

1 Introduction

In the future, software development will undergo a paradigm change. So far, hardware faults during execution were improbable and could be ignored. But as the integration of circuits approaches molecule transistors, hardware faults due to faulty production or external influences at runtime have become more and more likely [Sem11]. Transient faults, that means faults that only last for a short time, are especially critical because there is no way to predict them. Also, it is impossible to provide a workaround on hardware level, as for certain permanent circuit faults, because the unit that is affected, for instance by cosmic radiation, is not technically defunct, but still provides a wrong result.

There are several well-approved methods to protect software against faulty hardware behaviour; Jim Gray already suggested transaction-managed persistent processes in his 1985 paper [Gra85], along with other methods such as checkpointing that are still in use today. However, each layer of replication adds costly overhead, so it is important to identify the parts of the system that are essential and need to provide uninterrupted service, and protect only them. To isolate a part of the software and tolerate the failure of the other, the software system needs to be separated into modular components [Gra85]. One of the advantages of microkernel-based operating systems is that they provide software isolation at a low level by offloading almost all tasks into userland.

Still, an operating system needs to rely on a functional system kernel. Thus, the kernel is the most important part of the system, which is why we need to ensure its stable operation by all possible means. And because a fault confessed is half redressed, the first step to improve reliability is to find the regions in the software that are most sensitive to the consequences of transient hardware faults.

Therefore, I wrote a framework that enables the programmer to do fault injection on arbitrary target programs. My priority during development was to test the operating system kernel, which is harder than testing userland programs. Most importantly, a special virtual machine is required. I chose FailBochs for this task, which was developed by Schirmeier et al. specifically to test arbitrary system platforms, real or emulated hardware, on every level of the system [Sch+12]. I am going to evaluate this fault injection framework in the course of this work.

On top of FailBochs, I developed a custom experiment suite featuring four types of fault injection experiments: IDCFlip, RATFlip, ALUInstrFlip and GPRFlip. I am going to explain those experiments in detail and present the fault injection campaign I ran using them. Because the source code of the target software was available to me, I was able to reproduce individual experiment runs. I will present eight of those runs and give a detailed report on the respective hardware fault and its consequences to the system.

2 Motivation and State of the Art

This work deals with software fault injection, introducing a custom injection framework for fault injection in the operating system kernel. First, I need to define the terminology that is necessary to understand the remainder of this work. Then, I introduce the system I want to test. After that, I talk about fault injection and explain why I used it and what it is exactly. Then, I mention existing solutions and explain which framework I chose as a base for my experiments. Lastly, I deal with related work on software fault injection in the operating system kernel.

2.1 Fault, Error, Failure

This work aims to improve the *dependability* of a computer system, especially of the system's operating system kernel. Improving dependability means reducing the *failure rate* of a system. To be clear on what *failure* is and what causes it, we need to define the terms *fault* and *error* first.

According to Avizienis et al., "a system may fail either because it does not comply with the specification, or because the specification did not adequately describe its function" [ALR01]. They call the cause of this failure an error or, in their words, "a failure occurs when an error reaches the service interface and alters the service" [ALR01]. Avizienis et al. go on to describe a fault as "the adjudged or hypothesised cause of an error. A fault is active when it produces an error; otherwise it is dormant" [ALR01]. So, in the most interesting case, an active fault inherent in the system causes an error that results in a misbehaviour of the interface, which means that the system failed. If that failure again induces a fault into the system, we get the so called "fundamental chain of dependability threats" (Figure 2.1): A fault is propagating through the several layers of the system until it reaches the user interface, and each failure starting from a wrong logic decision at the gate level can cause a fault in the higher layers.

In case a fault caused a failure, it is essential for the system's dependability to find the fault and make sure that it cannot disturb the system in future runs.

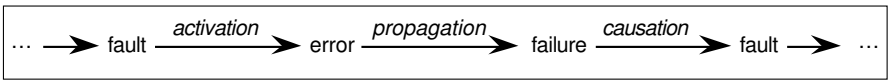

Figure 2.1: The chain of dependability threats [ALR01]

Not all kinds of faults can be detected using one solution. Applying the terminology of Avizienis et al., my research work treats only operational, transient hardware faults that are induced naturally from outside the system. These faults are called *single-event upsets* (SEUs), because they disturb the system's run at a single point in time and do not have permanent effects. Prominent examples for environmental influences on the system that cause these kinds of faults are cosmic radiation or electromagnetic background noise (environmental radiation) [ZL79].

2.2 Target System

The system that I am going to check for its dependability is our operating system group's research operating system, consisting of Fiasco.OC [Grob] and the L4 Runtime Environment (L4Re) [Groa]. This system is microkernel-based; if you want to learn more on contemporary microkernels, Jochen Liedtke's work provides information on their architecture and the functionality. [Lie95]

To ensure compatibility with legacy programs written for macrokernel operating systems, microkernel-based operating systems provide a so-called runtime environment. This framework offers backend servers that replace the kernel calls of a monolithic system. L4Re also contains an adapted standard C library that provides a front-end to these servers. Using this system setup, developers can access library functions like `malloc()` the way they are used to and do not have to rewrite their applications entirely when porting them to Fiasco.OC and L4Re.

Considering which components need to be tested, the microkernel is of special importance when analysing a system's dependability, because the microkernel provides the fundamental mechanisms that all other applications rely on, so we constantly try to improve its dependability.

One approach to ensure kernel dependability is to prove the correctness of the kernel code by proving theorems describing its expected behaviour. The most prominent microkernel project applying this method is seL4 [Kle+09]. They use a kernel that is further stripped to a required minimal amount of code that can be examined by theorem proving software. Klein et al. call this kind of stripped-down operating system core a "third-generation microkernel". Using several intermediate layers and a formal specification of the C language, they can prove that their source code is correct [Kle+09]. There are two problems with the seL4 approach: for one, it does not take into account any mistakes in the proof itself, which is conducted manually [Kle+09]. Also, Klein et al. mention that their proof stops at source code level, which means they "assume at least the compiler and the hardware to be correct" [Kle+09]. As I assume that the hardware exhibits faulty behaviour, the seL4 approach is irrelevant for my work.

If we have no theoretical means to improve a system's reliability, what remains is to examine its reliability practically, which means analysing the system while it is running. In the course of this work, I present means to simulate faulty behaviour deliberately at specific locations during a normal run of the target system and then check for its reaction.

For completeness, I should mention here that I used the Intel IA-32 version of Fiasco.OC. This instruction set architecture is better known by the code names of its associated processors as the x86 architecture. Intel's manual [Corb] provides a detailed documentation of its behaviour.

2.3 Fault Injection

As I already said in Section 2.2, I will test the system's dependability by running specific workload programs and modifying their state at an arbitrary location in space and time. This process is called *fault injection*.

The technique is far from new: The first publication I found to mention fault injection dates back to 1967, introducing the Saturn Fault Simulator [HS67]. The article treats hardware fault injection at the gate level, which is still in use today. Also, the paper already distinguishes

between transient (*intermittent* in their words) and permanent (*solid* in their words) faults in the hardware, and allows multiple faults to be injected in one run.

Today, according to Hsueh et al., fault injection experiments differ in their target's development state (*simulation-based fault injection* versus *prototype-based fault injection*, especially for hardware fault injection) as well as its position in the system's hierarchy (for instance *hardware fault injection*, applying logical or electrical faults, versus *software fault injection*, corrupting code or data). Simulation-based fault injection modifies a model of the actual system, which is useful to validate fault tolerance mechanisms included in the system. On the other hand, prototype-based fault injection delivers much more realistic results, because with that approach, the developer operates on a real prototype of the system using debugging interfaces to alter code, data or the flow of electric current [HTI97].

Among the methods available for fault injection, I had to choose one appropriate for the experiments I had in mind. The first consideration for this choice was how realistic the results of the experiments would be. As said in Section 2.1, my aim is to improve the reliability of the tested software in case a hardware failure occurs. The most sensible option seems to simulate the real hardware and inject the fault there, which is exactly what hardware fault injection does. For instance, the simulator could invert the logical decision on a specific gate or flip a bit on either the input or output line of a functional unit, acting toward the software as if an SEU had happened there.

Unfortunately, our target systems are IA-32 compatible, which means that there is no hardware specification of the central processing unit (CPU) available to us. Because hardware fault injection either needs special hardware capable of debugging itself or a model of the hardware that is at least plausible, performing hardware fault injection experiments is simply impossible for us at the moment. Hence, we have to use software fault injection, changing code and data instead of logical hardware behaviour.

According to Hsueh et al., employing software fault injection is not necessarily a loss: The advantage of software fault injection over hardware fault injection is that software fault injection is more flexible: Experiments do not require the hardware to cooperate. Using emulators, it is possible to test software for every computing architecture. Still, the fault injection framework cannot simulate faults in parts of the hardware invisible to the software. Also, you need to be careful to ensure that the actual injection does not disturb the run of the software under test [HTI97].

Consistent modelling is still rare among software systems. Because there is no complete description for my target system either, I use prototype-based fault injection.

2.4 Existing Tool Sets

My goal is to perform software fault injection, hence I had to find a tool for this purpose. For that, I chose to rely on an existing solution. I will now go on to outline the landscape of software-implemented fault injection (SWIFI) tools to explain my choice.

Fault injection as such is a well-defined process, which is why the general procedure is the same for most of the tools: You choose an injection point and an error type, then you run your target software up to that point. There, you perform one or more state manipulations,

and continue the program. If there is a reason to stop the execution, such as a timeout, or the program has finished, you collect the results of the experiment and quit.

Because of their algorithmic similarity, software fault injection frameworks mainly differ in their target and their design pattern. To choose one framework to start with, I took a closer look at these two criteria.

First, I encountered that there are two different starting points when designing a fault injection framework: One group of developers started from the target platform of their software and added fault injection features, whereas the others developed a fault injection framework independently of what they were about to test.

Schirmeier et al. describe these two categories of software fault injection solutions as *specialists* and *generalists*: Specialists provide a much broader access to the emulated hardware, yet in turn, they are directly attached to and sometimes mingled with their target. Generalists stand out due to their abstract, generally applicable software design patterns, which is why the computing platform can be replaced more easily here. On the other hand, generalists have fewer opportunities to manipulate the workload [Sch+12].

There are more examples for specialists, ranging from the more traditional, yet still influential FERRARI [KKA95] to the more recent FAUMachine [PSC07]. These two also comprise a wide field of design and concept: FERRARI is a user-level tool set specifically designed for fault injection that runs on an unmodified UNIX-compatible operating system, whereas FAUMachine is a fully featured IA-32 virtual machine that includes the ability to inject faults merely as one of its features. FIAT [Seg+88], on the other hand, is a generalist: It provides the users with a unique interface to describe their fault injection experiments. When a user has defined an experiment, FIAT passes it on to the target program, using an additional fault injection library layer in the software. GOOFI [Aid+01], as a more recent example, provides a similarly abstract design pattern. As you can see in Figure 2.2, the target system and the simulator are strictly separated from the fault injection framework.

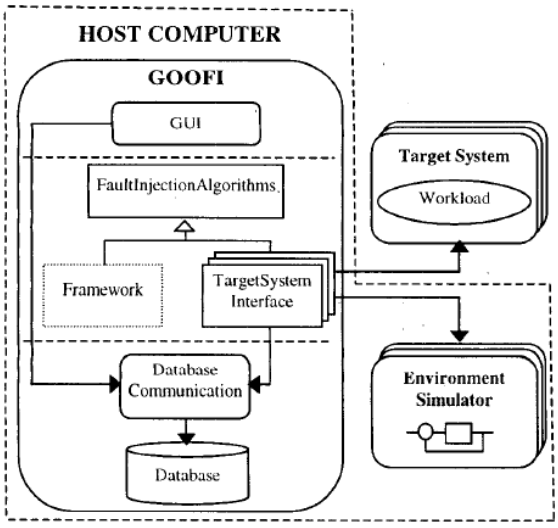

Figure 2.2: The GOOFI architecture [Aid+01]

The question is now whether it is possible to use the advantages of both the specialist and the generalist approach: I would like to perform manipulations in all stages of the emulation process and still keep the emulator exchangeable.

It turns out that the limits of the two approaches originate in their programming paradigms: Generalists encapsulate the individual parts of the injection framework into objects, so they have to integrate the emulator as a single object or an independent set of objects. If a generalist framework adheres strictly to the object-oriented paradigm, it needs to leave the emulator untouched, thus missing out on the internal functionality of the emulator. A specialist extends the emulator or the target program with functions that manage the fault injection. Because the specialist mingles fault injection and program functionality, it is hard to separate fault injection and emulation again in case the program changes.

It turns out that we can successfully close the gap if we consider the integration of an emulator system a different, cross-cutting concern and design the integration layer independent of the emulator as well as the fault injection environment.

FAIL*, the framework I based my experiments on, achieves that aim using *aspect-oriented programming* (AOP). In contrast to object-oriented programming, "AOP is based on the idea that computer systems are better programmed by separately specifying the various concerns (properties or areas of interest) of a system and some description of their relationships" [EFB01]. In a fault injection environment, planning and conducting a fault injection campaign and modifying the regular run of the system emulator are two different concerns. Consequently, FAIL* comprises two separate implementations: It contains a framework in regular `C++` with an abstract object-oriented design pattern, and a separate set of `C++` code that applies the faults to the emulator [Sch+12]. The `AspectC++` compiler then *interweaves* the two parts of the framework at specified *join points*. `AspectC++` is a `C++` dialect with syntactic elements that allow the programmer to insert aspect code in the right spots [SGSP02].

Because of the intermediate layer between framework code and emulator that is independent of both, in theory, the developer does not need to change the emulator any more to inject faults. Practically, a few hints as to where to attach the join points as well as minor state-saving additions are necessary, but in the large scale, the emulator remains untouched. In conclusion, it is possible to gain both flexibility and access to all necessary functionality, using the intermediate layer based on aspects. Because FAIL* features such a software architecture, it seemed to be a good starting point. Also, FAIL* had started as FailBochs, so from the beginning I could expect support for the IA-32 architecture. Because FAIL* continues to add new emulator backends, it will be possible to support other platforms our operating system runs on, such as ARM, so that they may be target of fault injection campaigns as well.

Now that I have presented the essential explanations necessary to understand the following chapters, I go on to describe how I constructed the experiment infrastructure in Chapter 3. Based on this framework, I designed and implemented several experiments, which I explain in Chapter 4. Furthermore, I evaluate the results of a fault injection campaign I conducted using the newly developed experiment clients in Chapter 5.

2.5 Related Work

This work is not the first to treat kernel fault injection. When talking about software fault injection, there are always faults injected in userspace that affect the kernel as well. Li et al., for example, discuss the effect of those faults issued in the application that propagate to the operating system kernel and corrupt its state [Li+08]

Considering dependability analysis of microkernels in particular, the articles on MAFALDA provide a deep insight into fault injection in a microkernel-based system. Arlat et al. divide the microkernel into separate components, such as "synchronisation, scheduling, memory, communication", which they then target individually using specifically tailored workload programs. Of these programs, each stresses one particular functional unit of the microkernel. For every fault injection campaign, one of the functional units is selected. While MAFALDA is running, it injects faults in either the parameters of calls to the kernel's application binary interface (ABI) or the code and data in the microkernel itself [Arl+02]. Arlat et al., too, assume a transient fault model, which I am going to explain in further detail in Section 3.2.

However, Arlat et al. focus on evaluating the conducted experiments and leave their readers but with a general description of the employed methods, partly because the kernel and the workload under test are proprietary and we cannot examine them at source code level. To compensate for their lack of transparency, I will present selected pieces of source code that are affected by a certain fault and behave accordingly in Section 5.2. For one, these examples will prove the correctness of the experiment: given the fault and the affected lines of the source code, the results should be predictable among the software I use. Secondly, the source code examples illustrate how we can achieve a higher level of dependability: A software developer can monitor if and how the target program reacts to hardware faults. Because each experiment run is deterministic, the developer can trace back the software failure unambiguously to the hardware fault. Using that kind of information, programmers are able to identify the regions of source code that are prone to a certain kind of fault and can focus on improving these particular code sections.

3 Construction of the Experiment Framework

There are two important aspects when conducting fault injection experiments: the target system and the parts of that system which are to be examined. In the course of this chapter, I will explain what requirements there were for either and how they will be fulfilled in the course of this work.

3.1 Requirements for the Target Platform

Elaborating the conditions outlined in Chapter 2, I came up with the following list of requirements for the injection target:

1) I am going to conduct software fault injection on an operating system kernel (see Section 2.2). For a first test of my experiment framework, I will use conventional userland applications.

2) The injection shall not interfere with the examined workload to maintain a realistic behaviour of the system (see Section 2.3).

3) As an addition to point 2, I require that if no injection was conducted, all experiment runs produce the exact same result. This postulate guarantees that if a system failure occurs, the error that causes the system to fail originates in the injected fault.

For point 1, I use the Fiasco.OC kernel and its userland L4Re in their IA-32 flavour to run the workload applications. I do not treat the 64-bit extension of IA-32.

If we strictly follow point 2 and allow no altering of the original software whatsoever, recompiling the software to insert fault injection functionality, as FIAT does, is already fundamentally wrong. Altering the program flow may not be problematic as long as the software is not reflecting on its own code, which is true for most userland applications. However, King et al. examined the reaction of operating systems to classic software debugging efforts, such as altering and displaying the operating system's state [KDC05]. They found out that "many aspects of operating systems make them difficult to debug: they are non-deterministic; they run for long periods of time; the act of debugging may perturb their state; and they interact directly with hardware devices" [KDC05]. From these results, I drew the same conclusions as King et al. and decided not to modify the operating system to inject faults. Hence, I needed to emulate the real hardware and then change the emulator in such a way that it could present the fault in the hardware to the operating system and capture its reaction.

One of the earliest backends implemented in FAIL* was one for the IA-32 emulator Bochs. I chose Bochs [Dev] over other emulators mainly because it merely relies on logical time: Bochs

starts off with an initial timestamp and inserts a timer interrupt every x instructions to adapt the system time, where x is the number of instructions per second configured in the Bochs configuration file divided by the frequency of the timer interrupt. That means that every execution path is deterministic because every source of pseudo-non-determinism commonly used in a computer, such as schedulers and random number generators, relies on the system clock. This determinism causes the workload to generate a constant output, and every deviation has to be caused by a fault injected into the program, which validates point 3.

My assumptions about Bochs' behaviour apply as long as no real time is required. One scenario may be that the workload program has to interact over a network with neighbouring computers that are ignorant of the application being subject of a fault injection campaign. Another example would be testing a device driver, because external hardware almost always relies on correct timing on the controlling computer. For these applications, it is hard to construct a plausible mechanism that guarantees reproducible results. A promising approach is called *reverse debugging*: the operating system is run in a modified, so called *time-travelling virtual machine* that includes advanced logging facilities, so that the specific run can later be replayed [KDC05]. The modified virtual machine can then simulate erroneous behaviour during the replay, which enables the developer to see what would have happened to the operating system if a fault had disturbed its run.

3.2 The Injection Scheme

The second important aspect of a fault injection framework is its fault model. I already explained that this work treats transient faults in Section 2.1 and why I use software fault injection in Section 2.3. Now I am going to provide details on the way I tried to model hardware behaviour in software to conduct experiments that are closer to reality than random bit flips.

As said in Section 2.5, MAFALDA uses a fault model that abstracts from the hardware in use, simulating corrupted bytes in arbitrary locations of kernel call parameters and code and data segments of the kernel. The way the data got corrupted is irrelevant, all that matters is whether the fault propagates to the system interface and causes the system to fail [Arl+02]. Chen et al. use a similar fault model for hardware-induced faults when testing the Rio file cache [Che+96].

The other radical approach would be to analyse the circuitry of the underlying hardware considering the behaviour of its basic blocks, if not of all its logical gates, and then create a fault model in software that exactly mirrors this circuit's properties during the fault injection in software.

Because such a model of an IA-32 processor, despite not being available to us, would probably exceed the boundaries of this work, I chose a compromise: the faults are injected at arbitrary instructions in the software or in the kernel, but their structure roughly follows the structure of a real hardware fault. For instance, when modifying the program data, I distinguish between a fault where the bus arbiter exchanges two registers (RATFlip) and a simple bitflip in a single register (GPRFlip), because these two faults are also independent on real hardware. This fault injection method implies the risk of misunderstanding the hardware and is as such always incomplete, at least as long as the algorithms are not based on a real hardware description. The approach is similar to the Text and NOP fault patterns applied by Swift et al. [Swi+06].

3.3 Selected Hardware Units

Having chosen to study the operation of the hardware, we now need to know which units in particular shall be considered faulty. For that, it might be worth to take a look at hardware fault injection. As a starting point, I followed Li et al., who examine eight functional units of a computer regarding their fault tolerance [Li+08]:

- The Instruction Decoder (IDC): While reading the instruction, before the IDC starts decoding, one bit of the instruction is flipped.
- The Integer Arithmetic Logic Unit (ALU): The result of an integer calculation differs by one bit.
- The Register Bus: While writing to or reading from a register, one bit is flipped on the bus.
- The Integer General Purpose Registers (GPR): One of the physical registers (or in our case, one of the architectural registers) suffers a bit flip.
- The Reorder Buffer: The out-of-order execution unit of the CPU exchanges one destination or one source register of one instruction in the reorder buffer with another register. This experiment is conceptually similar to the RAT experiments.
- The Register Alias Table (RAT): This table assigns logical, architectural registers to physical registers, and in the fault injection experiments, one of these assignments goes wrong.
- The Address Generation Unit: A bit is inverted in the virtual address used for a memory access.
- The Floating Point Arithmetic Logic Unit (FP ALU): Like in the Integer ALU case, one calculation result has one of its bits flipped.

Of these eight, I chose four as an initial feature set: the IDC, the integer ALU, the RAT and the GPRs. This selection matches with the fault injection experiments that have been carried out by our research group so far [DHE12]. In the following I provide reasons why I skipped the other half for the time being.

Because a fault in the reorder buffer has similar consequences compared to a fault in the RAT when seen from a software perspective, separating these to experiments will only be necessary when the exactness of the RAT injection algorithm has improved significantly.

Injecting faults in the floating point ALU is conceptually similar to manipulating the integer ALU, so I consider the earlier sufficient for a start.

Faults in a register and on the register buffer only differ in their persistence: while a fault that occurs directly in the storage area of the processor remains there due to the properties of the storage latches, even if it is not permanent, a transient fault on the register buffer only persists when writing to the register — a corrupted read affects only the current instruction. This difference is quite small, so I chose to implement only one variety.

Fault injection in the address generation unit for IA-32 basically means filtering the GPR experiments for registers bearing addresses, because IA-32 uses indirect addressing instead of

load-store instructions to access the main memory [Corb]. Although this approach sounds interesting, especially for upcoming RISC architectures, I postponed it due to the unstructured nature of IA-32, which makes programming this filter extremely difficult.

4 Design and Implementation of the Experiments

In the following chapter, I am going to provide an overview on how the experiments were designed and implemented. I divided the chapter into three parts: first, I want to talk about the improvements that had to be made to the FAIL* framework itself. Then, I explain the experiment algorithm in general, followed by a detailed description of the individual experiments.

4.1 Necessary Improvements of FAIL*

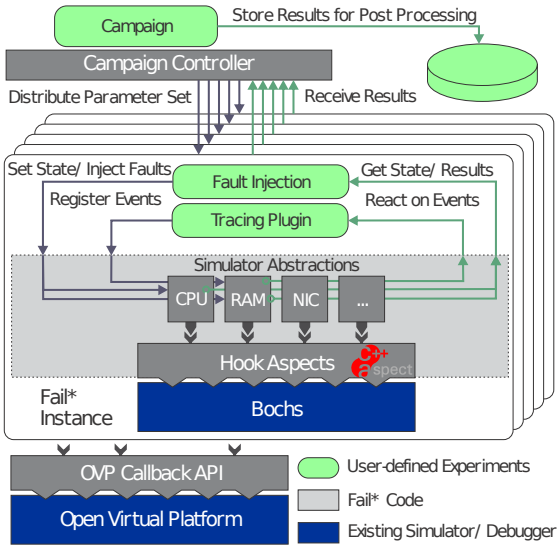

Figure 4.1: An overview on FAIL*'s software architecture [Sch+12]

To understand the improvements I made, I need to explain the FAIL* framework in further detail. I already said in Section 2.4 that FAIL* comprises two parts: the platform-independent framework code and the emulator-specific code that adapts the framework to the respective backends.

The interesting part for a software engineer is the framework code. As you can see in the overview of the software architecture in Figure 4.1, the experiment clients (labelled *Fault Injection* in the scheme) manipulate the emulator backend to simulate faulty hardware behaviour. The *Campaign Controller* sends experiment parameters to one or more experiment clients and is responsible for collecting and storing the results of the fault injection campaign. The scheme also includes a reference to the attached backends and the aspect-oriented code that connects

them. Additionally, the experiment clients can use several plugins (the example here is the *Tracing Plugin*) to perform generic tasks using a common code base.

Three improvements of the FAIL* framework were necessary, so that I could conduct the experiments efficiently and correctly: I introduced address space support as well as an additional event listener for IA-32 I/O ports and I started to work on the efficiency of event handling. These improvements were done in the *Simulator Abstractions* area of the framework, which is located in the experiment client.

As the latter two improvements are a little more complicated, I explain them in detail in the following subsections.

4.1.1 IOPortListener

To validate the correctness of a program run, I need to trace the output of the workload software. The L4/Fiasco operating system uses the EIA-232 serial line to communicate its console input and output. Hence, to trace the output of the system, the L4Sys experiment client needs to use this serial line interface.

Although current instruction set architectures usually map device memory into the user's virtual address space to enable I/O communication (memory-mapped I/O), IA-32 still provides a mechanism called *I/O ports*. In conclusion, to record the output that the emulated system generated on the serial line, I needed a dedicated event handling mechanism for these ports.

I/O ports are controlled using the `IN` and `OUT` instructions. The handler I wrote checks for the execution of one of those instructions and forwards the transmitted data to the experiment client.

4.1.2 Efficiency of the event handling mechanism

As you can see in Figure 4.1, FAIL* is largely event-driven. An excerpt of the event handling mechanism is displayed in the UML class diagram in Figure 4.2. Internally, when registering an event, the experiment client, which is basically an instance of `ExperimentFlow`, appends a `Listener` object to the internal event listener queue of the `SimulatorController` object. The `SimulatorController` class has only instance per experiment client, which represents the state of the attached emulator. The `SimulatorController` object has an instance of `ListenerManager` that manages its event queue. When the emulator reaches the state described by the event listener, the aspect layer that connects the emulator to the fault injection framework returns the control flow to the `ExperimentFlow` object that issued the listener. To do so, the adaption layer uses methods of the `SimulatorController` object, such as `onBreakpoint()` or `onInterrupt()`, to find the appropriate listeners for the current event. The now active experiment client can evaluate and alter the current state of the executed workload. After all operations on the emulator are done, the experiment client registers the next listener and returns the control flow to the emulator (using either the `addListenerAndResume()` or the `resume()` method). If there is nothing more to do, the experiment client quits and can optionally terminate the emulator, too. The `ListenerManager` stores all registered event listeners in a common queue (`m_BufferList`).

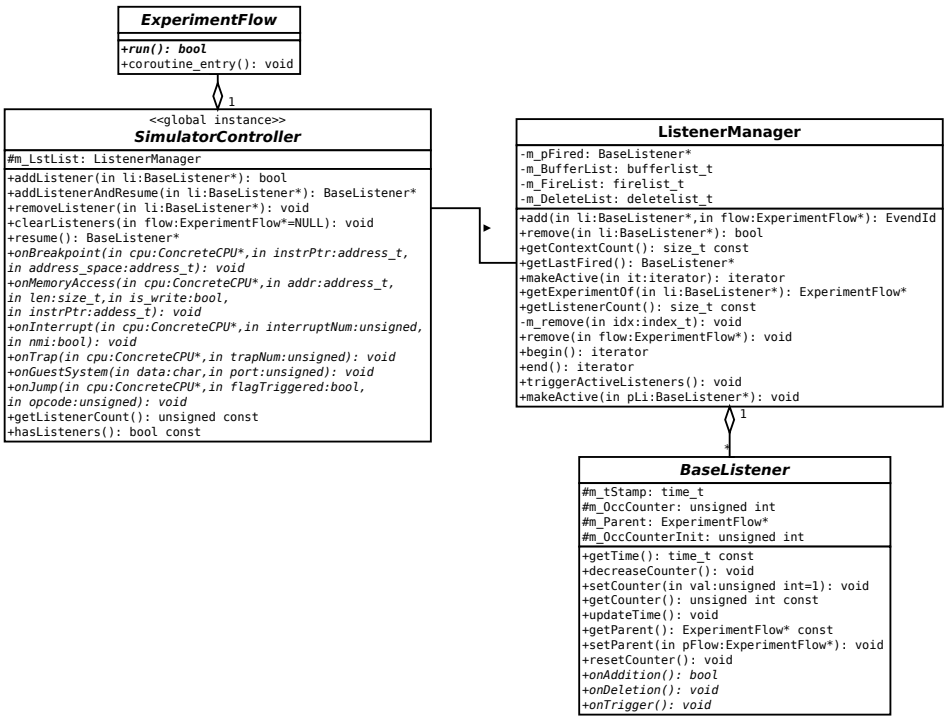

Figure 4.2: UML class diagram describing the FAIL* event handling mechanism (simplified)

The algorithm I just briefly outlined tries to solve a problem that is two-dimensional: At a specific event in the emulator, only a subset of listeners can possibly be affected. For instance, whether or not the instruction pointer of the emulated CPU has changed is only relevant for triggering breakpoint listeners. In a unified queue, the ListenerManager has to store the listeners merely as BaseListener objects to guarantee the extensibility of the event class tree. Thus, when using one queue for all types of event listener objects, the ListenerManager needs to find out for each element in the queue whether it has the right type or not at runtime.

Consequently, to check if the current event requires to trigger a certain listener, all listener objects in ListenerManager are checked for their type using `C++`'s `dynamic_cast` feature and, if the type matches, further tests are applied. The overhead of this implementation is humanly perceptible even for small experiments, so I thought of closing the performance gap by caching.

In the solution I came up with, the event handler function selects the relevant subclass by using the appropriate `BufferCache` object. The `BufferCache` supplies wrappers for all methods in `ListenerManager` that are necessary to iterate and activate event listeners, and replaces the previously applied dynamic casting by static casting via templates.

There used to be instances of the BufferCache template for the two most frequently used subclasses of BaseListener: BPListener, which is checked every time the emulator changes the instruction pointer, and the IOPortListener. The latter is only frequent in my experiments, because they verify the program run using its output on the serial interface (see Section 4.1.1).

Recently, the mechanism was reimplemented as an optional feature using aspects, which enables the developer to compare the speed of the simulation with the feature enabled and disabled, respectively. The new implementation fits better into the framework object structure, however, it is now specifically tied to one event type, so in the future, event types need to be added manually. From the performance point of view, the two solutions are about equal in case the compiler optimisations are enabled. Using my BufferCache implementation, an exemplary measurement of the execution time of 25 GPRFlip experiments showed a speedup of roughly 2.5, while a recent measurement of the new implementation using the *hsc-simple* experiment showed a speedup of about 2. For corner cases, a speedup of up to 7.5 could be achieved [Böc12]. A downside of the new solution is that it slows down the framework if the compiler does not remove the enlarged object structure, which was not the case for my implementation.

4.2 The Experiment Process

This section outlines the common procedure for all four experiments. They only differ in the fault pattern they apply during the modification phase of the experiment.

4.2.1 Initiation of an experiment

The FAIL* framework transmits experiment data via a socket-based client-server system using Google's Protocol Buffer technology, *Protobuf* in short [Goo], which provides a convenient way to transfer data over a network independent of the utilised platform and programming language. In this model, the campaign server (labelled *Campaign Controller* in Figure 4.1) loops over all fault injection scenarios requested by the user, puts each of them in a packet and sends the packet to one or more experiment clients, which perform the actual fault injection.

Each packet transmitted by the server tells the client what kind of experiment to conduct, and at which instruction what kind of data needs to be changed. I am going to explain that mechanism in detail now.

4.2.2 Adjusting experiment parameters

The aim of this work is to provide a reusable infrastructure for injecting several types of faults into simulated hardware. I implemented this infrastructure using the dynamically configurable *L4Sys experiment client*. The experiment type, the instruction and bit offset as well as an identifier for the affected register, if necessary, are provided by the campaign server for every single experiment at runtime. These parameters are called *dynamic parameters*. There are also parameters that are *static*. These parameters are stored in a special header file and only change when the client is recompiled. Examples include the entrance and exit addresses of the code under test, the number of executed lines of code and the target program's address space.

Before it can start running, L4Sys needs to transmit the following dynamic parameters:

- The *instruction offset* specifies before which instruction the fault shall occur, counted from the beginning of the target application.
- The *bit offset* describes which bit to modify within the injection target.

- The *experiment type* selects an individual injection pattern within L4Sys.

4.2.3 Performing the experiment

As soon as the client has received the data necessary to start the experiment, it applies the selected fault injection pattern.

To inject the fault, Bochs executes from a predefined state up to the start instruction. As soon as it has reached the first instruction the emulator switches back to the experiment code, where the fault is applied as requested, before switching back to the target system. The program subsequently continues to run until it finishes, or the framework detects a timeout.

The result is returned to the campaign server, including the absolute address of the faulty instruction. In case an error occurred, the client adds detailed information that may be useful for finding the weak spot in the tested workload, such as the last instruction pointer and the output generated by the user's application.

4.2.4 Evaluation of the experiment data

Using the newly gained information, the campaign server creates a chart listing the return values of the experiment clients.

In the default implementation, the data collected by the campaign server is stored using the comma-separated values (CSV) file format, with the numerical identifiers for the fields *experiment type*, *register type* and *result* automatically converted to their string equivalent. While this data format makes the files bigger, it also ensures they are easy to read for the person evaluating the results.

4.3 Types of Experiments

For each of the functional units under test listed in Section 3.3, I developed a specific fault injection experiment, which I integrated into L4Sys. Each of them can inject faults at arbitrary instructions of arbitrary workload programs.

4.3.1 IDCFlip

IDCFlip simulates a fault in the instruction decode latch, because of which the CPU decodes and executes an instruction with one bit inverted. Depending on the bit position, that bitflip either affects the processed data or the executed operation. Whether such a fault is likely to happen or even possible is hard to answer without hardware specifications: Given the complexity of current IA-32 instruction decoders, such a particular event may be improbable. Still, from a software point of view, the overall outcome would not differ much if the injection happened in another stage of instruction decoding.

To implement this experiment, we need direct access to the instruction cache of the emulator to read and modify the current instruction. Bochs uses a structure called `bxInstruction_c` to store information on an instruction. The only information relevant for this experiment is the length of the instruction, which is known to vary widely in the IA-32 architecture. Using this

information, the experiment client computes the modulo of the transmitted bit position and the size of the instruction to retrieve the position of the bit to invert.

In the next step, the client calculates the position of the current instruction using Bochs' internal methods. Then, the client reads the machine code from Bochs' main memory representation and inverts the target bit. Now the experiment client decodes and executes the newly generated instruction, again using Bochs' internal methods.

After the faulty instruction has been executed, the experiment client replaces it with the original structure that was backed up beforehand, so that the fault is transient and not persistent. This experiment uses singlestepping to ensure that only the modified instruction is executed before returning to the experiment client to restore the original instruction. For all experiments that use singlestepping, there is an additional scheduling timeout of 10 seconds. If the kernel has not switched back to the address space of the workload program within that time, the experiment client assumes that the system failed while executing the single modified instruction.

4.3.2 RATFlip

RATFlip features a simple approach to simulate an error during the association of instruction architecture registers to hardware registers. The basic assumption is that in the register alias table (RAT), a bit is inverted and causes the input or output, depending on the instruction, to be taken from or written to a random different register.

4.3.2.1 Disclaimer

RATFlip makes several assumptions: First, the experiment client acts as if only one thread were currently running, which is per se incorrect because we operate on a multitasking-only system. In a microkernel operating system, even basic services like memory management are part of the userland and run in separate threads that can break the whole system when damaged. Future versions of RATFlip should come up with a sufficient listing of all threads running on the system and randomly distribute faults among their register sets, with the remaining registers assumed empty.

Also, we model the registers that are not occupied by the current process' architectural registers as empty, which is not necessarily true either, because the results of computations are partially delivered via bypasses to avoid pipeline stalls. Examples include out-of-order execution, or register renaming as suggested by Tomasulo [AST67]. Temporary registers used by these algorithms contain important results, which may even be used by completely unrelated threads (see above).

Still, following these assumptions, we can elaborate a sufficiently simple fault model for the register allocation table (by Döbel et al. [DHE12]): with a likelihood of 10 percent, the fault hits another register of the thread, and with likelihood of 90 percent, the data comes from or is written to an unused register. In the first case, random input data is generated, whereas in the second case, the output of the instruction is simply ignored. This relation originates from the hardware architecture of current Intel processors of the *Sandy Bridge* generation. While featuring 16 registers in its instruction set architecture (due to the 64-bit-extension of IA-32), these processors have 160 physical integer GPRs installed [Cora].

The worst problem with my current approach is that it does not associate input and output registers correctly for the respective IA-32 instructions. For now, I assume that the first register is always the output register, which is not true for a lot of instructions: CPUID, for instance, implicitly overwrites half the IA-32 register set. Here, a table featuring a set of an input, output and clobber registers per instruction will be necessary in the future.

4.3.2.2 Experiment process

First, to run the experiment, we need to check whether the current instruction uses registers. If it does not, we need to step over until we find an instruction that works with registers, and do the injection there. This procedure allows for arbitrary entry points into the program, which makes the experiment suitable for long programs because the experiment client does not need to know the instructions passed in advance, yet searching the appropriate instruction using singlestepping implies an additional overhead.

The experiment client then conducts a random 1 out of 10 decision, as described in the previous section. According to this decision, the target of the fault is either a GPR of the same process or an unused physical register.

In the first case, and if the source register is an input register, the contents of the two registers are simply swapped for the time the instruction is executed, and then swapped back afterwards. If the source register is an output register, the swapping happens after the execution of the faulty instruction. In the case of an unused machine register, the output is discarded and the input is randomly generated.

To execute the instruction under faulty conditions, this experiment also uses singlestepping, which means the scheduling timeout described above applies here, too.

4.3.3 ALUInstrFlip

Originally, when suggesting ALU fault injection, Li et al. thought of modifying the output of the ALU [Li+08]. For that, I would have needed a complete list of all output registers of all integer ALU instructions. Finding out about output and input registers in IA-32 is a general problem (see Section 4.3.2.1). Because I lacked the necessary information, neither input nor output value modification where an option, so I decided to change the instruction the ALU executed. The results would probably not differ much among the three tests, because what matters is that the computation produces a wrong result. As all faults are assumed to be transient, it is not important where the wrong value originated.

ALUInstrFlip resembles IDCFlip, but it is limited to instructions that are “generic” and either “arithmetic”, “logical” or perform a “shift and rotate” operation, according to the *geek32 edition* website [edi]. I associate these instruction categories with the tasks of an ALU. Please note that an actual ALU may not have a separate instruction decoder, and the set of instructions that I selected may not be exhaustive.

First, ALUInstr checks if the current instruction is included in the set of instructions mentioned above. If not, the experiment client steps through the target code until it finds an ALU instruction or reaches the end of the testing area (much like RATFlip, see Section 4.3.2.2).

When the experiment client has found an ALU instruction, the `bxInstruction_c` object describing the current instruction is converted into a custom `BochsALUInstr` structure. Each

`BochsALUInstr` objects belongs to one of currently twelve equivalence classes for IA-32 integer ALU instructions. Each of these equivalence classes represents a particular addressing mode, ignoring the use of registers.

Then, the experiment client selects one random ALU instruction with an equivalent addressing mode, so that it can reuse the current `bxInstruction_c` object, with slight modifications, and avoid faulty memory accesses due to uninitialised values therein.

Finally, the experiment client single-steps over the new instruction, which means the scheduling timeout applies here, too. After the experiment client has executed the faulty command it writes back the original instruction, which happens the same way as in Section 4.3.1.

4.3.4 GPRFlip

This experiment simulates a bit flip in an arbitrary integer general purpose register (GPR). In contrast to the other experiments, which work on synchronous latches, this experiment simulates memory, making the introduced faults persistent until they are overwritten again. Therefore, GPRFlip's algorithm is as simple as a single data manipulation: the content of the GPR is inverted in the requested bit position, and subsequently, the execution of the workload program is resumed.

4.4 Tested Workload

First, I had to make sure that my experiments worked correctly, so I started off with the straightforward Bitcount benchmark from the MiBench suite [Gut+01] to gain comprehensible results. The benchmark is mainly executed in userland, only 4.3 percent of the instructions are executed in kernel space. The benchmark was run 100000 times, which resulted in an overall of 83082714 executed instructions.

The main target of the L4Sys experiment client is the microkernel, thus it was essential that my experiments could perform reliable kernel-level fault injection. To stress the kernel in particular, I used the pingpong benchmark of our group as the workload program for this second fault injection campaign. I chose the "IPC inter AS" benchmark (short for "inter process communication between different address spaces"), which creates two processes that subsequently exchange messages with each other. The benchmark was executed ten times, which means 80 rounds of sending a message back and forth. An overall 7877560 instructions where executed, 7684300 of them in kernel space (98 percent).

5 Evaluation

The experiments presented in this chapter shall demonstrate how the L4Sys experiment works and that it can be used to analyse the dependability of its target software. First, I introduce the types of results that an experiment can return. Using these result types, I present several model experiments and describe what happens there in detail. Finally, I evaluate all the experiments I ran, including their execution time statistics.

5.1 Result Types

Once an experiment is finished, its outcome can be categorised according to the failure levels mentioned by Saggese et al. [Sag+05]: *No effect*, *Crash*, *Incomplete execution* and *Silent data corruption*. I will now explain each possible exit reason in a few words to clarify the terminology.

- *No effect*: Although a fault was injected into the emulated hardware, it does not manifest in the result the program delivers. The software may not read a faulty value, e.g. in a specific GPR, before it is overwritten again, or the result of a failed calculation may still be sufficiently accurate for the current purpose. Also, according to Wang et al., about one third of all branch commands are outcome-tolerant, meaning that no matter which path you choose, the program flow converges back to the same point [WFP03].
- *Crash*: The program finishes with an abnormal exception. In my model, crashes manifest as timeouts in the FAIL* framework. Saggese et al. generally assume a ten percent overhead [Sag+05], which I deemed reasonable in this case, too.
- *Incomplete execution*: The program exceeds its normal execution time, in terms of executed instructions, and is consequently terminated. The cause for this misbehaviour is usually an endless loop caused by the injected fault, but a chain of wrong branch decisions is imaginable as well. The infinite loop may as well be the input loop of an operating system exception handler that is never responded to, because most Fiasco.OC exception handlers let the user enter the kernel debugger by pressing a key. The latter case is also a *Crash* and thus equivalent to that result type. To detect incomplete execution, Saggese et al. suggest to wait until the execution time of the program is exceeded by ten percent [Sag+05]. Here, again, I decided to follow their recommendation.
- *Silent data corruption*: The program completes in time without crashing, but returns a wrong result. If there is no replicated run of the software that can be used for comparison, there is no way to detect this kind of fault. However, in a fault injection environment with the necessary prerequisites, each deviance in the program output has to be caused by the injection (as described in Section 3.1).

5.2 Selected Experiments

In this section, I am going to describe eight specific experiment runs. I selected these runs from my experiment set so that each experiment type and each result type are represented once for my kernel and my userland benchmark, respectively. An overview of this selection scheme can be seen in Table 5.1.

This case study demonstrates how specific errors manifest in program state and provides a deeper understanding of the ways in which hardware faults may appear during execution.

Result / Experiment	No effect	Crash	Incomplete execution	Silent data corruption
GPRFlip	U	K		
RATFlip			U	K
IDCFlip	K	U		
ALUInstrFlip			K	U

Table 5.1: The selected experiments that are discussed in detail

5.2.1 Experiments in userland

I start with the userland experiments to convey the general idea of my fault injection experiments using simple error scenarios. To understand what faults in the emulator can cause which errors in the software, we first need an overview on the piece of code that is affected.

5.2.1.1 The target function

Consider the following piece of C code from the Bitcount benchmark:

```
1  int CDECL AR_btbl_bitcount(long int x)
2  {
3        unsigned char * Ptr = (unsigned char *) &x;
4        int Accu;
5
6        Accu  = bits[ *Ptr++ ];
7        Accu += bits[ *Ptr++ ];
8        Accu += bits[ *Ptr++ ];
9        Accu += bits[ *Ptr ];
10       return Accu;
11 }
```

This function counts the bits that are set to one in a 4-byte `long int` value. To do so, it uses a lookup table called `bits`, which is defined according to the following scheme:

```
static char bits[256] =
{
      0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,  /* 0   - 15  */
      1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,  /* 16  - 31  */
      1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,  /* 32  - 47  */
      ...
      4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8   /* 240 - 255 */
```

```
};
```

This table stores the number of bits set to one in a byte that contains a specific value between 0 and 255. In the table, each counter value also takes up one byte.

For the actual counting, the function stores the address of the long value in a bytewise addressed pointer (line 3). Then, it accumulates the bit count for each byte in the variable Accu, using the unsigned value of the current byte as an index into the lookup table (lines 6-9). When the function has processed all four bytes, it returns Accu as its result (line 10).

To make my experiments more comprehensible, I chose to generate the code without compiler optimisations for this case study. The piece of C code discussed previously translates to the following assembler code:

```
0x0100177f <AR_btbl_bitcount>:
# line 2
 0x100177f       push   ebp
 0x1001780       mov    ebp,esp
 0x1001782       sub    esp,0x10
# line 3
 0x1001785       lea    eax,[ebp+0x8]
 0x1001788       mov    DWORD PTR [ebp-0x4],eax
# line 6
 0x100178b       mov    eax,DWORD PTR [ebp-0x4]
 0x100178e       mov    al,BYTE PTR [eax]
 0x1001790       and    eax,0xff
 0x1001795       mov    al,BYTE PTR [eax+0x101e040]
 0x100179b       movsx  eax,al
 0x100179e       mov    DWORD PTR [ebp-0x8],eax
 0x10017a1       inc    DWORD PTR [ebp-0x4]
# line 7
 0x10017a4       mov    eax,DWORD PTR [ebp-0x4]
 0x10017a7       mov    al,BYTE PTR [eax]
 0x10017a9       and    eax,0xff
 0x10017ae       mov    al,BYTE PTR [eax+0x101e040]
 0x10017b4       movsx  eax,al
 0x10017b7       add    DWORD PTR [ebp-0x8],eax
 0x10017ba       inc    DWORD PTR [ebp-0x4]
# line 8
 0x10017bd       mov    eax,DWORD PTR [ebp-0x4]
 0x10017c0       mov    al,BYTE PTR [eax]
 0x10017c2       and    eax,0xff
 0x10017c7       mov    al,BYTE PTR [eax+0x101e040]
 0x10017cd       movsx  eax,al
 0x10017d0       add    DWORD PTR [ebp-0x8],eax
 0x10017d3       inc    DWORD PTR [ebp-0x4]
# line 9
 0x10017d6       mov    eax,DWORD PTR [ebp-0x4]
 0x10017d9       mov    al,BYTE PTR [eax]
 0x10017db       and    eax,0xff
 0x10017e0       mov    al,BYTE PTR [eax+0x101e040]
 0x10017e6       movsx  eax,al
 0x10017e9       add    DWORD PTR [ebp-0x8],eax
# line 10
 0x10017ec       mov    eax,DWORD PTR [ebp-0x8]
 0x10017ef       leave
```

```
0x10017f0       ret
0x10017f1       nop
0x10017f2       nop
0x10017f3       nop
```

The three nop operations at the bottom are used for memory alignment; they are never invoked. I preceded the remaining instructions with the corresponding line numbers. If an explanation is necessary for understanding the respective program failure, I explain the specific functionality of the assembler code in the following sections.

5.2.1.2 Experiment 1: GPRFlip affecting EDI at 0x10017d0, Result *No effect*

The register EDI is not used in this ADD operation, nor in any other instruction that follows, as Bitcount does not use repeated load/store/test operations, and none of the system functions that do are invoked after the injection. Therefore, the program runs until the end and produces a correct result.

5.2.1.3 Experiment 2: IDCFlip at 0x10017ae, Result *Crash*

In this experiment, the following assembler instruction is affected:

```
0x10017ae       mov    al,BYTE PTR [eax+0x101e040]
```

The injection turns the machine code instruction 0x8a 0x80 0x40 0xe0 0x01 0x01 into 0x9a 0x80 0x40 0xe0 0x01 0x01 0xXX, turning a move instruction into a far call [edi]. XX stands for a random byte that the decoder reads from main memory because by specification the far call instruction is one byte longer than the MOVE instruction. The resulting assembler instruction looks like this:

```
0x10017ae       call   0xXX01:0x1e04080
```

Far calls access another memory segment to do a function call. In this case, the program tried to execute code beyond its segment limit, which causes a general protection fault in the hardware. For the protection fault, Bochs produces the following error message:

```
[CPU0 ] fetch_raw_descriptor: GDT: index (f07) 1e0 [exceeds] limit (67)
```

GDT is for *global descriptor table*, describing among others the segments and their limits (see Intel's IA-32 Manual, Volume 3 [Corb]). The operating system is notified of the protection fault and subsequently informs the program. The application can now provide the user with details about its failure before the operating system terminates it. Unfortunately, the fault handling routine provided by L4Re states merely that there had been an "unhandled exception", but omits what fault caused this exception.

5.2.1.4 Experiment 3: ALUInstrFlip at 0x10017d0, Result *Silent data corruption*

At this point, another bit count should be added to Accu:

```
0x10017d0       add    DWORD PTR [ebp-0x8],eax
```

Instead, the ALU decides to negate Accu:

```
0x10017d0       neg    DWORD PTR [ebp-0x8]
```

Because one of the summands is now negative, the overall bit count for the current method ("Non-recursive bit count by bytes (AR)") that is calculated in the benchmark's main loop is too low (1472887 instead of 1472906).

5.2.1.5 Experiment 4: RATFlip at 0xf003e1c3, Result *Incomplete execution*

This experiment represents a fault that occurs in kernel space and propagates to userland. The fault occurs relatively late, such that only the last line of the program is not printed. This failure happens because of an endless loop caused by an incorrect restore to EDI in the return section of the `Mux_console::write` function:

```
f003e1be:       pop    edx
f003e1bf:       mov    eax,ebp
f003e1c1:       pop    ebx
f003e1c2:       pop    esi
f003e1c3:       pop    edi
f003e1c4:       pop    ebp
f003e1c5:       ret
```

This register is often used to store a temporary stack pointer during function calls, which means that in a nested function call, the pointer to the stack frame is moved into an invalid memory area. Here, the program accidentally continues to run in a valid code section and executes an endless loop.

5.2.2 Experiments in kernel space

For the kernel space experiments, I chose to present four different pieces of code to examine a larger range of kernel code functionality, and also because I want to use this section to present the rather rare result cases, such as an instruction flip that has no effect or a general purpose register bit flip that leads to a crash. As those occur rarely anyway, and because kernel functions are usually short, it is improbable to find all these types of failures within one complete `C++` kernel function.

Because of that, the structure differs from the previous section; I present a piece of code for each experiment, and then explain what effect the fault had on that program segment.

5.2.2.1 Experiment 5: IDCFlip at 0xf0015e5d, Result *No effect*

Consider the following kernel function:

```
1  PROTECTED inline
2 void
3 Receiver::set_rcv_regs(Syscall_frame* regs)
4 {
5   _rcv_regs = regs;
6 }
```

This piece of code does a simple pointer assignment, resulting in the following assembler instruction:

```
0xf0015e5d       mov    DWORD PTR [ebx+0xf8],eax
```

Now, its binary representation has been modified from `0x89 0x83 0xf8 0x00 0x00 0x00` to `0x88 0x83 0xf8 0x00 0x00 0x00`, making it an 8-bit operation:

```
0xf0015e5d       mov    BYTE PTR [ebx+0xf8],al
```

Reducing this pointer operation to the first byte should make the operating system crash while accessing an invalid memory region. Fortunately, the target memory cell and the register already contain equal values, so it does not matter how much data is copied. So as you can see, in addition to branch commands (see Section 5.1), data manipulation instructions are also often outcome-tolerant.

5.2.2.2 Experiment 6: GPRFlip affecting ESP at 0xf0023268, Result *Crash*

At this address, we find the binary representation of the following Assembler code:

```
0xf0023268       call thread_timer_interrupt /* enter with disabled irqs */
0xf002326d       pop    edx
0xf002326e       pop    ecx
0xf002326f       pop    eax
0xf0023270       iret
```

In this experiment, the stack pointer is moved up two memory pages, which means that it still points into a valid memory region, but at completely wrong data. So instead of returning from a timer interrupt handler to the normal program flow, the subsequent IRET instruction reads the target code segment descriptor as `NULL`, as can be seen in the according Bochs error message:

```
00216385463e[CPU0 ] iret: return CS selector null
```

`NULL` is an invalid segment identifier, so the operating system issues a general protection fault. While the kernel tries to figure out what went wrong, it fails too because its own assumptions on the layout of the stack are now also wrong. The kernel page fault combined with the protection fault results in a double fault and thus a failure of the whole operating system. Considering the severe consequences of such a single bitflip in one register, it may be sensible to increase the dependability of future hardware by protecting essential registers like the stack pointer or the instruction pointer using additional redundancy.

5.2.2.3 Experiment 7: ALUInstrFlip at 0xf001a0ff, Result *Incomplete execution*

The error that causes the benchmark to fail occurs within the memory map operation. Consider this piece of C++ code:

```
1  L4_error __attribute__((nonnull(1, 3)))
2  mem_map(Space *from, L4_fpage const &fp_from,
3          Space *to, L4_fpage const &fp_to, L4_msg_item control)
4  {
5    typedef Map_traits<Mem_space> Mt;
6
7    [...]
8
9    // loop variables
10   Mt::Addr rcv_addr = fp_to.mem_address();
11   Mt::Addr snd_addr = fp_from.mem_address();
12   Mt::Addr offs = Virt_addr(control.address());
```

```
13
14  Mt::Size snd_size = Mt::Size::from_shift(fp_from.order() -
15                      L4_fpage::Mem_addr::Shift);
16  Mt::Size rcv_size = Mt::Size::from_shift(fp_to.order() -
17                      L4_fpage::Mem_addr::Shift);
18
19  // calc size in bytes from power of twos
20  snd_addr = snd_addr.trunc(snd_size);
21  rcv_addr = rcv_addr.trunc(rcv_size);
```

The fault occurs while truncating the sender address:

```
1  Target trunc(Target const &size) const
2  { return Target(_v & ~(size._v - 1)); }
```

In assembler, the affected part of the inlined `trunc` function consists of the following instructions:

```
0xf001a0f7       mov    ebp,DWORD PTR [esp+0x34]
0xf001a0fb       mov    ecx,DWORD PTR [esp+0x30]
0xf001a0ff       and    ecx,DWORD PTR [esp+0x2c]
0xf001a103       and    ebp,edx
```

Instead of performing the first AND operation, the emulator subtracts the memory value from register ECX, which results in a size of 0xFFFFFFFF instead of 0x0000102d for the sent page. Consequently, the kernel reports an invalid mapping:

```
KERNEL: Warning: nothing mapped: (Mem_space) from [0xfc4acaac/31]:
ffffffff size: 00000001 to [0xfc4aca0c/4a]
```

Due to the fault, the program runs slightly over time and transmits 46 units less than normal.

5.2.2.4 Experiment 8: RATFlip at 0xf003f470, Result *Silent data corruption*

This hardware fault appears in the `memset` function of the small libc implementation that the Fiasco microkernel includes (called *minilibc*).

```
1 void * memset(void * dst, int s, size_t count) {
2     register char * a = dst;
3     count++;    /* this actually creates smaller code than using count-- */
4     while (--count)
5         *a++ = s;
6     return dst;
7 }
```

In line five, register EBX and thus the pointer is not incremented. These assembler instructions correspond to line 5:

```
0xf003f46d       mov    BYTE PTR [eax+ebx*1],dl
0xf003f470       inc    ebx
```

Consequently, the kernel memory is not overwritten completely, and even this small fault generates an erroneous value that the kernel keeps using. The error subsequently causes a completely wrong output. Note that more efficient code is not always a dependability risk: using `count--` above, as suggested in the comment, would have resulted in a DEC instruction in place of the increment, which is just as vulnerable to the applied fault. On the other hand,

if the overall number of instructions is greater, it is harder to hit one instruction in particular, which may be a fault-sensitive one.

5.3 Evaluation of the Conducted Experiments

The experiments described in the following sections shall demonstrate that my framework functions correctly. They are in no way statistically relevant, because their source code coverage is only 0.006 percent for the Bitcount experiment and 0.07 percent for Pingpong per injection scheme.

The biggest difference when comparing the results of the Pingpong campaign in Figure 5.6 with those of the Bitcount campaign (Figure 5.1) is that the portion of faults resulting in a *Silent data corruption* is a lot smaller among kernel fault injection experiments. One of the reasons is that kernel code has a lower level of programming and is thus more vulnerable. Also, the kernel performs tasks that are highly critical, especially in a microkernel operating system. If such an essential function fails, unlike in userland, it is often not possible to finish the program with a wrong result. Instead, the program, several programs or the whole system fail immediately.

One example for increased kernel code vulnerability is a bitflip in a random register: because the code is highly optimised, partly also written in assembler, compared to userland applications, it uses a lot more registers at a time and the content of a specific register changes more frequently. Therefore, if the experiment client picks a random register and alters its content, this fault is more likely to cause a serious problem in kernel space than it is in userland.

Another reason is the level of optimisation applied: While I disabled compiler optimisation for Bitcount to generate more plausible assembler code, I compiled the kernel with the second-highest optimisation level of the `gcc C++` compiler, which is `-O2`. The generated code is dense, with a correspondingly high register reusing rate (see above).

The execution times of the four experiment types are nearly normally distributed for both campaigns (Figures 5.2, 5.3, 5.4 and 5.5 for Bitcount and Figures 5.7, 5.8 and 5.10 for Pingpong), with the exception of the Pingpong IDCFlip experiments (Figure 5.9). An explanation for the first peak in the graph could be that IDCFlip experiments induce completely random instructions into the program. Those instructions are more likely to induce a processor halt in Bochs because they simply cannot be interpreted. A dedicated scheduling timeout detects whether the injected instruction already caused the system to fail and if so, the experiment client skips the rest of the program, causing fewer computational overhead (see Section 4.3.1). Thus, the actual execution time of the experiment is a lot lower. Please note that compared to the other Pingpong timing charts, the Y scale of the IDCFlip chart is twice as high. However, this effect becomes negligible for larger programs: for Bitcount, the overall execution time is largely determined by the program execution time. In the chart, the execution times are sampled in intervals of 250 milliseconds for Bitcount and 62.5 milliseconds for Pingpong.

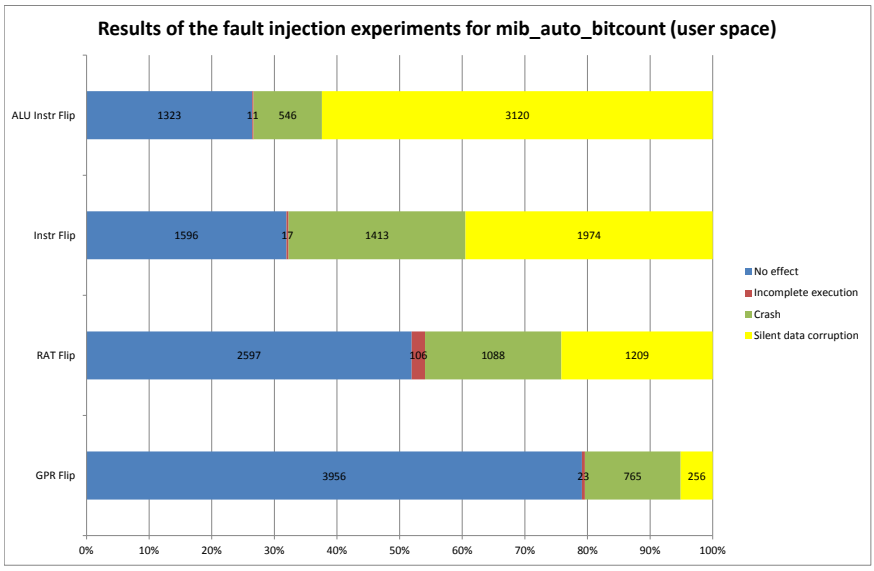

Figure 5.1: Results of the Bitcount campaign

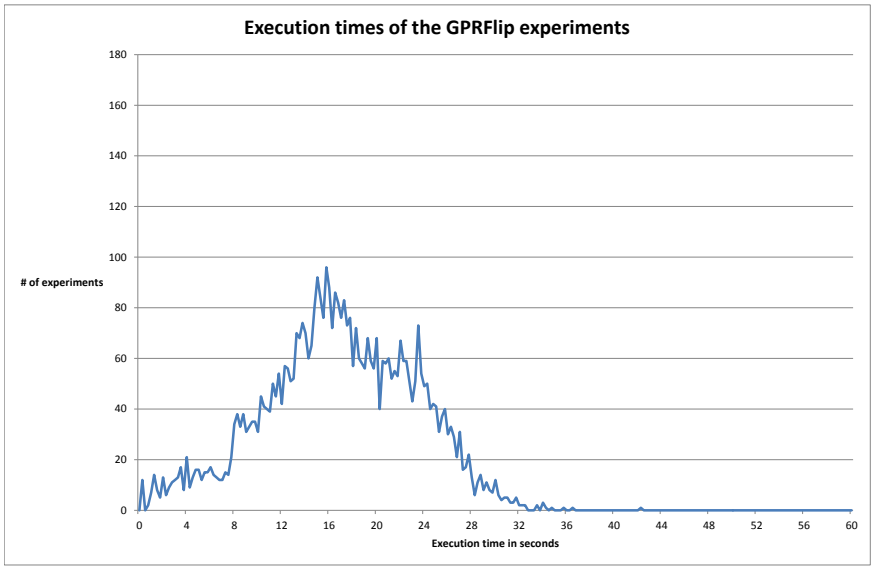

Figure 5.2: Execution times of the GPRFlip experiments

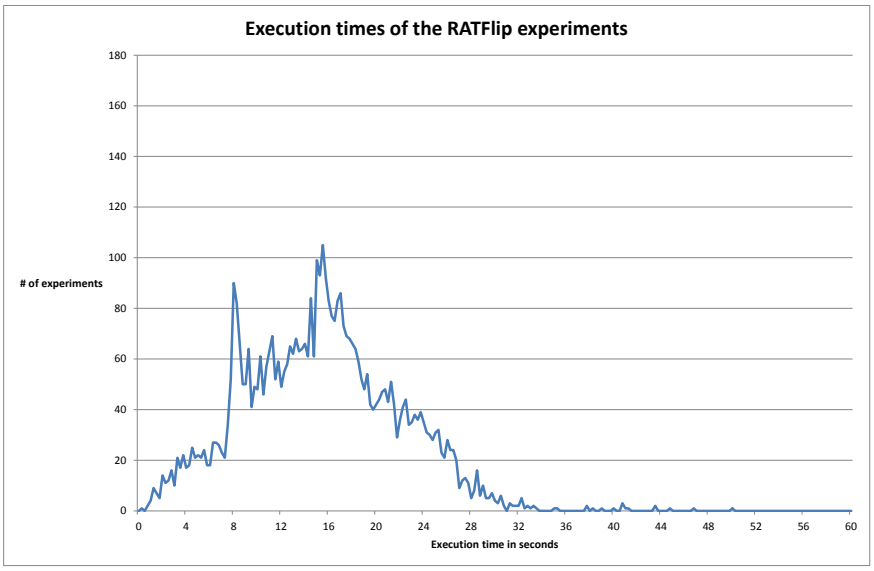

Figure 5.3: Execution times of the RATFlip experiments

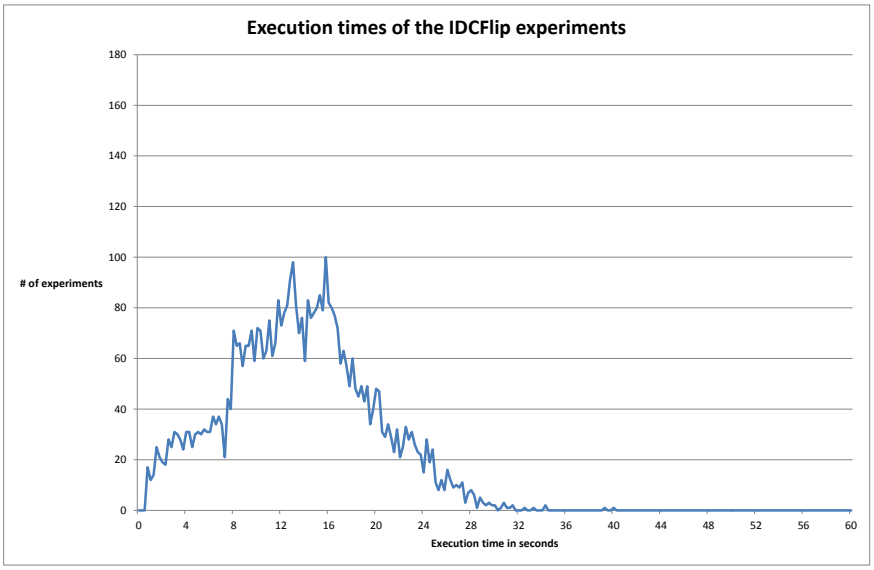

Figure 5.4: Execution times of the IDCFlip experiments

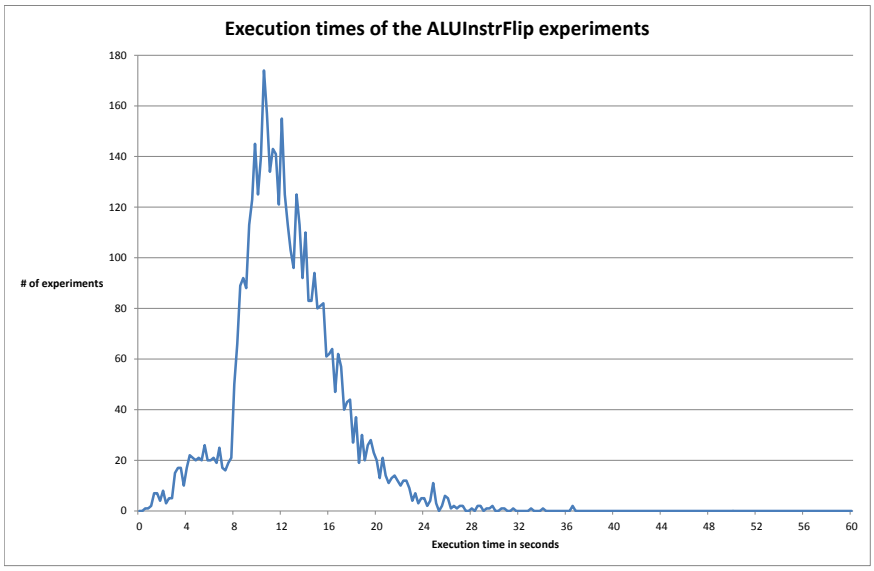

Figure 5.5: Execution times of the ALUInstrFlip experiments

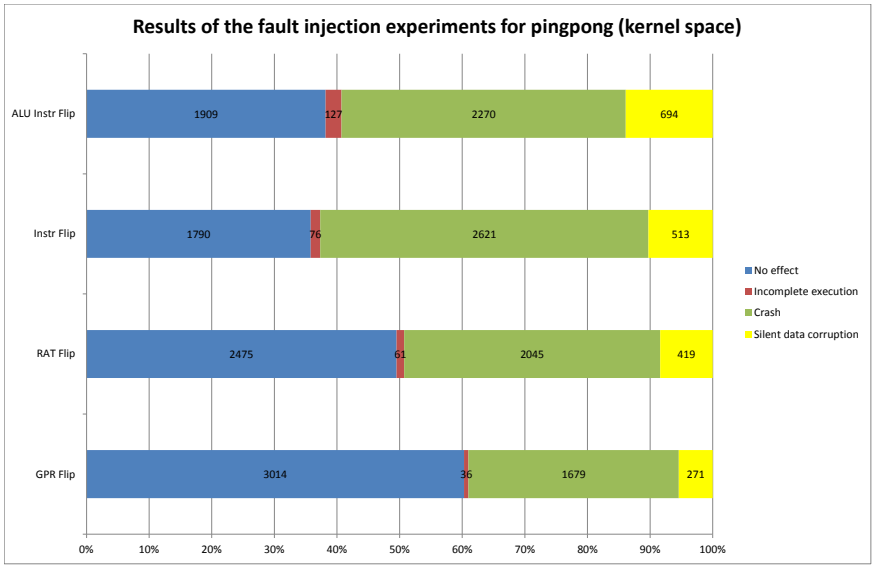

Figure 5.6: Results of the Pingpong campaign

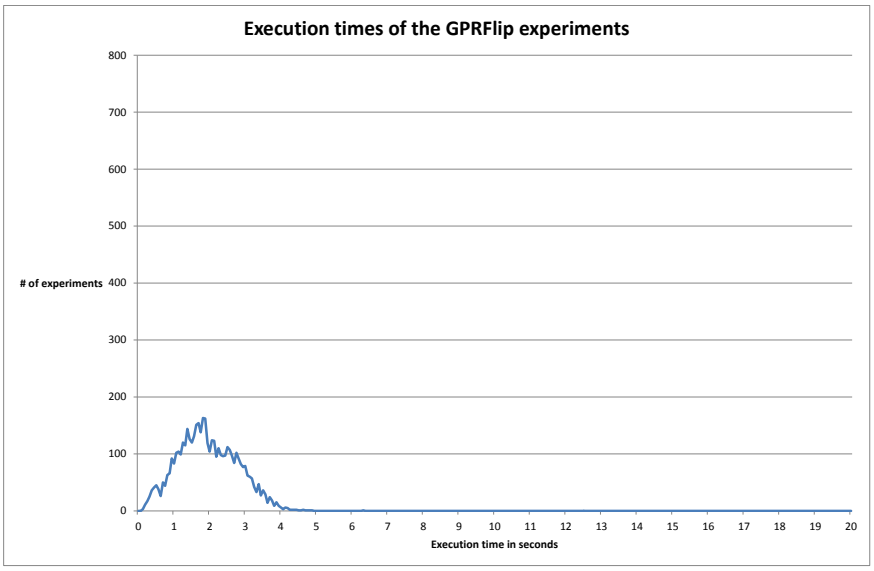

Figure 5.7: Execution times of the GPRFlip experiments

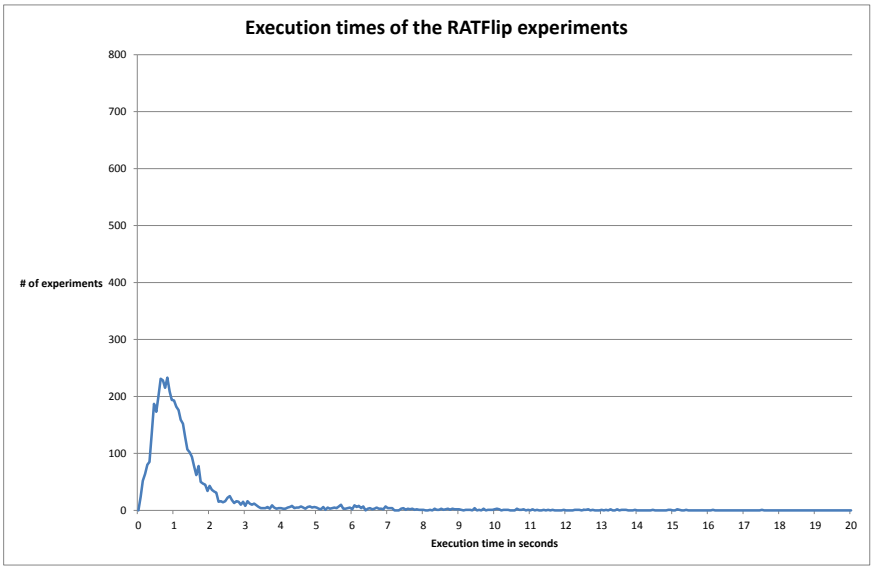

Figure 5.8: Execution times of the RATFlip experiments

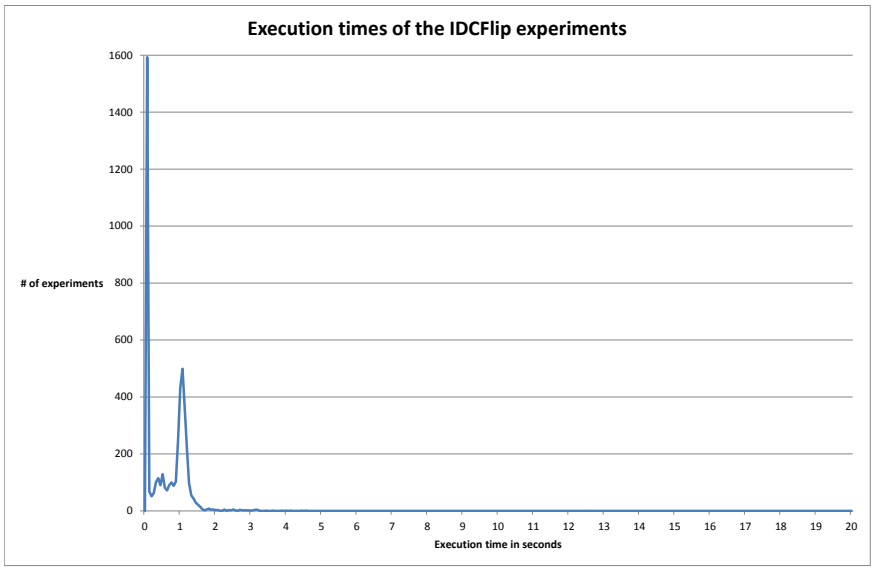

Figure 5.9: Execution times of the IDCFlip experiments

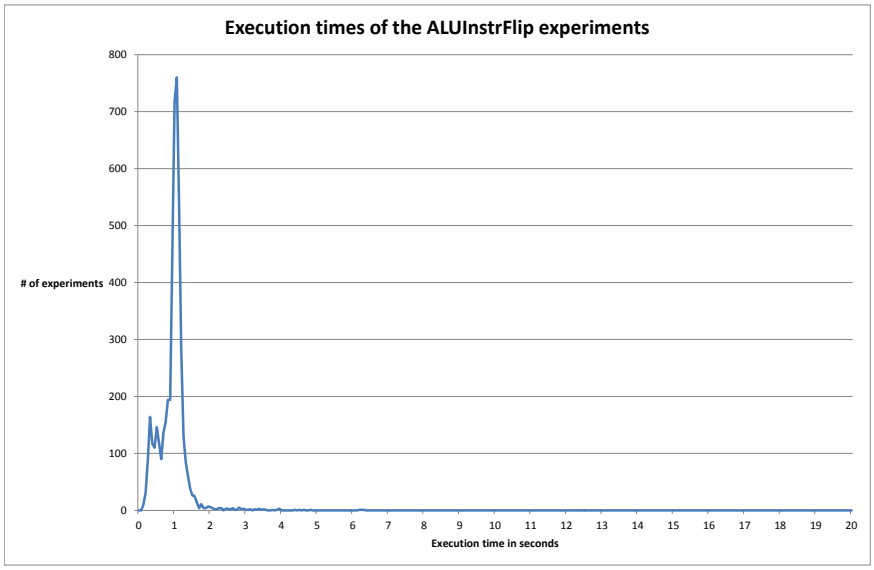

Figure 5.10: Execution times of the ALUInstrFlip experiments

6 Conclusion

6.1 Summary

Fault injection already is an important part of software test systems, and its importance will grow in the future as the hardware manufacturing size decreases and the exact reasons for transient as well as permanent hardware faults will be increasingly hard to find.

FAIL* is a fault injection framework that is well suited to perform kernel space fault injection. As I outlined in Section 3.1, FAIL* provides all means necessary for controlled fault injection, plus the possibility to exchange the target architecture.

As for the experiment types I selected, I can state that the choice I made is representative regarding the treated hardware units and the fault patterns. The fault injection experiments can be used to simulate faults of the hardware that affect code as well as data of the target system in a manner that is close to real hardware faults. Therefore, the results are better suited for software dependability analysis than those of simple byte flipping injections (see Section 3.2).

The experiments I conducted to test my framework showed that it is often hard to track the fault that caused the system to fail, for instance if the data is corrupted in a fail-silent manner. Those kinds of failures can only be detected using special techniques that I mentioned in the introduction, like process replication and checkpointing. Additionally, the emulator behaviour may differ from that of the real hardware: Faults may occur that would slip on the actual implemented hardware, for instance if the emulator strictly adheres to the specification and the hardware implementation does not (see Chapter 5).

6.2 Future Developments

For future versions of the framework, it will be necessary to become more independent of a specific emulator. At the moment, the ALUInstrFlip and the IDCFlip experiments are still tied to Bochs' procedures and data structures, as you can see in the descriptions in Section 4.3. Using Bochs, which is an IA-32-only emulator, it will not be possible to extend the experiments toward the promising field of RISC architectures, such as ARM, which rapidly continue to gain importance in future computing devices. To achieve independence from Bochs, the FAIL* framework will need additional features, like an assembler and a disassembler, and the program code must be adapted to those new features. Also, because Bochs relies only on logical time, a new model for timing will be necessary. This model should provide real-time capabilities, enabling the programmer to test the software under completely new scenarios, but it should still maintain fully reproducible experiment runs.

Subsequently, new experiment types should be implemented: The experiments I selected in Section 3.3 give a rough idea of what is possible, yet Li et al. mention four more hardware units that can be modelled [Li+08]. Implementing experiment clients that target those hardware units

would allow for an even more fine-grained fault injection campaign. Still, it is also important to constantly improve and update the hardware model of the existing clients: new, more detailed insight on the actual hardware implementation should immediately be integrated into the existing experiments' code. At this point, I should refer to the RATFlip experiment, which is still implemented in quite a fuzzy manner (see Section 4.3.2.1).

Fault injection is a time-consuming enterprise that is often criticised for delivering few results compared to the complexity of the job. Hence, developers that advocate fault injection aim at improving the ratio of complexity and insight by injecting faults more sensibly, that means only in regions of the system where they may cause a failure.

One means to measure the vulnerability of certain parts of a program to a specific fault is the so-called *program vulnerability factor* (PVF) introduced by Sridharan and Kaeli [SK09]. This factor is an architecture-independent measure, enabling analysis of fault propagation on program code level. In contrast, most of the other metrics in the field, such as the *architectural vulnerability factor* (AVF), describe the vulnerability of certain elements of the hardware architecture. Because the PVF is independent of the hardware, it is a good starting point for speeding up software fault injection campaigns, as Döbel et al. showed in their article treating an actual implementation of the concept [DSE13].

Bibliography

[ALR01] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. *Fundamental concepts of dependability*. 2001.

[AST67] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. “The IBM System/360 Model 91: Machine philosophy and instruction-handling”. In: *IBM Journal of Research and Development* 11.1 (Jan. 1967), pp. 8 –24. ISSN: 0018-8646. DOI: 10.1147/rd.111.0008.

[Aid+01] J. Aidemark et al. “GOOFI: generic object-oriented fault injection tool”. In: *International Conference on Dependable Systems and Networks (DSN 2001)*. July 2001, pp. 83 –88. DOI: 10.1109/DSN.2001.941394.

[Arl+02] Jean Arlat et al. “Dependability of COTS microkernel-based systems”. In: *IEEE Transactions on Computers* 51.2 (Feb. 2002), pp. 138 –163. ISSN: 0018-9340. DOI: 10.1109/12.980005.

[Böc12] Adrian Böckenkamp. “Measurements of the Fast Breakpoints extension”. Personal communication. Oct. 2012.

[Che+96] Peter M. Chen et al. “The Rio file cache: surviving operating system crashes”. In: *SIGOPS Operating Systems Review* 30.5 (Sept. 1996), pp. 74–83. ISSN: 0163-5980. DOI: 10.1145/248208.237154.

[Cora] Intel Corporation, ed. *2nd Generation Intel Core Processor Family Desktop Datasheet, Volume 1*. URL: http://www.intel.com/content/www/us/en/processors/core/2nd-gen-core-desktop-vol-1-datasheet.html (visited on 12/01/2012).

[Corb] Intel Corporation, ed. *Intel 64 and IA-32 architectures software developer’s manual*. URL: http://download.intel.com/products/processor/manual/325462.pdf (visited on 09/01/2012).

[DHE12] Björn Döbel, Hermann Härtig, and Michael Engel. “Operating system support for redundant multithreading”. In: *12th International Conference on Embedded Software (EMSOFT)*. Tampere, Finland, 2012.

[DSE13] Björn Döbel, Horst Schirmeier, and Michael Engel. “Investigating the limitations of PVF for realistic program vulnerability assessment”. In: *Proceedings of the 5th Workshop on Design for Reliability (DFR 2013)*. Berlin, Germany, Jan. 2013.

[Dev] The Bochs Developers. *Bochs: the open source IA-32 emulation project*. URL: http://bochs.sourceforge.net/ (visited on 09/01/2012).

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. “Aspect-oriented programming: Introduction”. In: *Communications of the ACM* 44.10 (Oct. 2001), pp. 29–32. ISSN: 0001-0782. DOI: 10.1145/383845.383853.

[Goo] Google. *Protocol Buffers*. URL: `http://code.google.com/p/protobuf/` (visited on 06/01/2012).

[Gra85] Jim Gray. *Why do computers stop and what can be done about it?* Tech. rep. 85.7. Tandem Computers, June 1985.

[Groa] Dresden Operating Systems Group. *L4Re — The L4 runtime environment*. URL: `http://os.inf.tu-dresden.de/L4Re/` (visited on 09/01/2012).

[Grob] Dresden Operating Systems Group. *The Fiasco microkernel*. URL: `http://os.inf.tu-dresden.de/fiasco/` (visited on 09/01/2012).

[Gut+01] M.R. Guthaus et al. "MiBench: A free, commercially representative embedded benchmark suite". In: *IEEE International Workshop on Workload Characterization (WWC-4. 2001)*. Dec. 2001, pp. 3 –14. DOI: `10.1109/WWC.2001.990739`.

[HS67] Fred H. Hardie and Robert J. Suhocki. "Design and use of fault simulation for Saturn computer design". In: *Electronic Computers, IEEE Transactions on* EC-16.4 (Aug. 1967), pp. 412 –429. ISSN: 0367-7508. DOI: `10.1109/PGEC.1967.264644`.

[HTI97] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. "Fault injection techniques and tools". In: *Computer* 30.4 (Apr. 1997), pp. 75 –82. ISSN: 0018-9162. DOI: `10.1109/2.585157`.

[KDC05] Samuel T. King, George W. Dunlap, and Peter M. Chen. "Debugging operating systems with time-traveling virtual machines". In: *Proceedings of the annual conference on USENIX*. Anaheim, CA, 2005. URL: `http://dl.acm.org/citation.cfm?id=1247360.1247361`.

[KKA95] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. "FERRARI: a flexible software-based fault and error injection system". In: *IEEE Transactions on Computers* 44.2 (Feb. 1995), pp. 248 –260. ISSN: 0018-9340. DOI: `10.1109/12.364536`.

[Kle+09] Gerwin Klein et al. "seL4: Formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: `10.1145/1629575.1629596`.

[Li+08] Man-Lap Li et al. "Understanding the propagation of hard errors to software and implications for resilient system design". In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2008.

[Lie95] Jochen Liedtke. "On micro-kernel construction". In: *SIGOPS Operating Systems Review* 29.5 (Dec. 1995), pp. 237–250. ISSN: 0163-5980. DOI: `10.1145/224057.224075`.

[PSC07] S. Potyra, V. Sieh, and M. Dal Cin. "Evaluating fault-tolerant system designs using FAUmachine". In: *Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems*. EFTS '07. Dubrovnik, Croatia, 2007. ISBN: 978-1-59593-725-4. DOI: `10.1145/1316550.1316559`.

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. “AspectC++: an aspect-oriented extension to the C++ programming language”. In: *Proceedings of the fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*. CRPIT ’02. Sydney, Australia, 2002, pp. 53–60. ISBN: 0-909925-88-7.

[SK09] V. Sridharan and D.R. Kaeli. “Eliminating microarchitectural dependency from architectural vulnerability”. In: *IEEE 15th International Symposium on High Performance Computer Architecture (HPCA 2009)*. Feb. 2009, pp. 117 –128. DOI: 10.1109/HPCA.2009.4798243.

[Sag+05] G.P. Saggese et al. “An experimental study of soft errors in microprocessors”. In: *IEEE Micro* 25.6 (Dec. 2005), pp. 30 –39. ISSN: 0272-1732. DOI: 10.1109/MM.2005.104.

[Sch+12] Horst Schirmeier et al. “Fail: towards a versatile fault-injection experiment framework”. In: *ARCS Workshops (ARCS), 2012*. Feb. 2012, pp. 1 –5.

[Seg+88] Z. Segall et al. “FIAT – Fault injection based automated testing environment”. In: *Digest of papers of the 18th International Symposium on Fault-Tolerant Computing (FTCS-18)*. June 1988, pp. 102 –107. DOI: 10.1109/FTCS.1988.5306.

[Sem11] International Technology Roadmap for Semiconductors. “Reliability”. In: *International Technology Roadmap for Semiconductors – Process integration, devices, and structures (PIDS)*. ITRS consortium, 2011. Chap. 6.

[Swi+06] Michael M. Swift et al. “Recovering device drivers”. In: *ACM Transactions on Computer Systems* 24.4 (Nov. 2006), pp. 333–360. ISSN: 0734-2071. DOI: 10.1145/1189256.1189257.

[WFP03] N. Wang, M. Fertig, and Sanjay Patel. “Y-branches: when you come to a fork in the road, take it”. In: *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT 2003)*. Oct. 2003, pp. 56 –66. DOI: 10.1109/PACT.2003.1238002.

[ZL79] J. F. Ziegler and W. A. Lanford. “Effect of cosmic rays on computer memories”. In: *Science* 206.4420 (Nov. 1979), pp. 776–788.

[edi] geek32 edition, ed. *X86 opcode and instruction reference 1.11*. URL: http://ref.x86asm.net/geek32.html (visited on 09/01/2012).