Revisiting Fault-Injection Experiment-Platform Architectures

Horst Schirmeier*, Martin Hoffmann†, Rüdiger Kapitza†, Daniel Lohmann† and Olaf Spinczyk*
*Department of Computer Science 12, Technische Universität Dortmund, Germany
e-mail: {horst.schirmeier, olaf.spinczyk}@tu-dortmund.de
†Department of Computer Science 4, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
e-mail: {hoffmann, rrkapitz, lohmann}@cs.fau.de

***Abstract*—Many years of research on dependable, fault-tolerant software systems yielded a myriad of tool implementations for vulnerability analysis and experimental validation of resilience measures. Trace recording and fault injection are among the core functionalities these tools provide for hardware debuggers or system simulators, partially including some means to automate larger experiment campaigns.**

We argue that current fault-injection tools are too highly specialized for specific hardware devices or simulators, and are developed in poorly modularized implementations impeding evolution and maintenance. In this article, we present a novel design approach for a fault-injection infrastructure that allows experimenting researchers to switch simulator or hardware backends with little effort, fosters experiment code reuse, and retains a high level of maintainability.

I. Introduction

Fault-injection (FI) experiments and dynamic trace analyses are common means to analyze a complex software-stack's susceptibility to hardware faults, and to assess the effectivity of software fault-tolerance measures [1]. Repeating the analysis/evaluation and software-hardening steps allows system designers to converge to an application-specific tradeoff eligible for their product. Over the last decades, a multitude of research projects, each with a different set of requirements, a zoo of hardware or CPU/system simulator platforms, and varying fault models and experiment setups, developed FI experiment tool suites [2]. Many of these tools incorporate some means to automate the process of repeating and experiments or varying fault models.

In this article, we partition the state of the art into two disjoint classes of FI tools. We argue that both classes entail significant disadvantages for the tools' users (regarding experiment code reusability, system-state access, or fault-model flexibility) and its developers (regarding software maintainability and evolution), and outline possible reasons for these downsides. Based on this analysis, we propose a novel architecture for a versatile FI experiment framework, based on the aspect-oriented programming (AOP) paradigm.

II. State of the Art

There exist FI experiment tools for dependability evaluations in different development phases, based on hardware simulators at varying levels of simulation accuracy, or on physical prototype hardware accessed through hardware-debugging interfaces. Existing FI tool implementations can roughly be partitioned into *generalists* and *specialists*: The former aim at high flexibility regarding FI target backend and portable experiment code, the latter specialize on a single target hardware/simulator and offer deep system-state access combined with variable fault models.

The *generalists* claim a certain level of flexibility regarding the target-platform backend. Among the benefits of this approach is that experiments can more easily be reused on a different platform—e.g., for gaining evidence the tested fault-tolerance measure is not platform-specific, or to move from a simulator backend to a real hardware prototype in later development phases. With GOOFI, Aidemark, Skarin et al. presented such a generic FI framework, abstracting away target systems in a plugin-based architecture [3], and additionally providing extensive pre- and post-experiment analysis methods. Fidalgo et al. [4] describe a generic tool addressing FI via the NEXUS on-chip debugger interface. Another example is QINJECT (David et al., [5]), injecting faults into a target backend utilizing the GDB debugger interface. These approaches have the common disadvantage that the chosen interface between experiment engine and target backend heavily limits access to target-system state, and narrows the possibilities for FI—e.g., obstructing the possibility to inject networking-device–specific faults into QEMU in the latter example.

In contrast, the *specialist* tools are highly specific to a single target. An example is FAUMACHINE (Sieh et al., [6]), which provides access to a large part of its x86 simulator's state, and enables various FI methods, including, e.g., hard-disk faults. But despite the advantage of providing access to the backend's full capabilities, this class of tools is characterized by severe maintainability issues: Deep state-access usually results in deep intrusion into the backend's code-base. Unfortunately, enhancing a simulator with FI code implemented in a traditional imperative language such as C or C++ often leads to intermixing the implementation of different *concerns*—in this case particularly the *simulation* and *fault-injection* concerns. Listing 1 illustrates this so-called *tangling* (different concerns implemented in a single implementation module) effect on a code example taken from FAUMACHINE;

This work was supported by the German Research Council (DFG) focus program SPP 1500 under grants KA 3171/2-1, LO 1719/1-1 and SP 968/5-1.

```
1  static int ide_gen_disk_read_raw(struct cpssp *cpssp,
2                                   uint8_t *buffer, uint32_t blkno) {
3    unsigned int i, defect;                                  Fault Injection
4    assert(blkno < cpssp->phys_linear + RESERVED_SECTORS);   Checks
5    for (i = 0; ; i++) {
6      if (i == sizeof(cpssp->fault) /
7               sizeof(cpssp->fault[0])) {
8        defect = 0;
9        break;
10     }
11     if (cpssp->fault[i].type == BLOCK
12      && cpssp->fault[i].blkno == blkno) {
13       defect = cpssp->fault[i].val;
14       break;
15   } }
16   switch (defect) {
17   case 0: /* No read error. */                            Fault Injection
18     storage_read(cpssp->media, buffer, 512, blkno * 512ULL);
19     return 1;                                  Normal Simulator Operation
20   case 1: /* ECC corrected read error. */
21     storage_read(cpssp->media, buffer, 512, blkno * 512ULL);
22     return 0;
23   case 2: /* un-correctable read error. */
24     memset(buffer, 0, 512);
25     return -1;                                           Fault Injection
26 } }
```

Listing 1. Code excerpt of FAUMACHINE's hard-disk simulation code (raw-block read): Implementation of *sanity check* (grey), *fault injection* (red) and *normal simulator operation* (green) concerns is heavily tangled.

```
1 aspect MemWriteHook {
2   pointcut mem_write() =                  pointcut: "where" to hook into Bochs
3     "void ...::bx_cpu_c::write_virtual_%(...)";
4   advice execution (mem_write()) : before () {
5     // divert control flow to FI module
6     fi::memwrite_trigger(tjp->arg<0>(),
7           tjp->arg<1>(), tjp->arg<2>());   advice: "what" to do there
8 } };
```

Listing 2. Aspect implementation of a hook diverting control flow from the BOCHS simulator into a module for fault injection on the data bus.

a related problem is called *scattering* (distribution of a concern implementation across multiple implementation artifacts). The resulting *tight coupling* between simulator and FI code often makes it difficult or even impossible to exchange the tool's target backend later on; in the case of tools that were forked from an existing hardware simulator, such as QEMU[1], even keeping in sync with the simulator's mainline evolution is often too arduous.

III. SEPARATION OF CONCERNS

When choosing an adequate compromise between *generalist* and *specialist* tools, one might very soon end up worsening the situation, combining the *disadvantages* of both worlds: A generic interface with little access to system state, implemented with tightly-coupled FI and backend modules. We believe that these problems are inevitable with traditional implementation approaches: Even if all FI-related code in listing 1 were modularized in a single function in a separate translation unit, the simulator would still have to be supplemented with a function call, passing control-flow to the FI implementation.

[1]For example, David et al. modified Qemu in [7], but seemingly later gave up on the fork in favor of a *generalist* [5].

We argue that only *strict separation of concerns* can avoid this maintenance nightmare. **Aspect-oriented programming** (AOP, [8]) is a technique known to facilitate this: So-called *aspects*—defining *where* ("pointcut") FI code (*what:* "advice") should be applied—allow for compact, well-encapsulated realizations of FI concerns. An "aspect weaver" automatically takes care of compile-time intermixing of simulator and extension code. Listing 2 exemplifies the approach, showing an AspectC++ [9] implementation of an aspect diverting control flow from the memory module[2] of the BOCHS simulator to a FI module, completely eliminating the need to invade the simulator's code manually. The join point API (see [9] for details) is leveraged to hand over (pointers to) the actual parameters to the `fi::memwrite_trigger` function.

We aim at proving the advantages of this approach by designing a new, versatile FI framework, based on off-the-shelf simulators such as BOCHS, QEMU, or OVP. We expect that loose target-backend coupling will significantly facilitate switching to a different backend, or synchronizing with newer backend versions. By additionally providing an API abstracting from backend commonalities, we intend to foster experiment code reuse, in summary combining the *advantages* of the two FI tool classes.

REFERENCES

[1] A. Benso and P. Prinetto, Eds., *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation (Frontiers in Electronic Testing)*, 1st ed. Boston: Springer, Oct. 2003.

[2] H. Ziade, R. A. Ayoubi, and R. Velazco, "A survey on fault injection techniques," *The International Arab Journal of Information Technology*, vol. 1, no. 2, Jul. 2004.

[3] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: Generic object-oriented fault injection tool," in *31st IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN '01)*. IEEE, 2001.

[4] A. Fidalgo, M. Gericota, G. Alves, and J. Ferreira, "Using NEXUS compliant debuggers for real time fault injection on microprocessors," in *Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design*. ACM, 2006.

[5] F. M. David, E. Chan, J. Carlyle, and R. H. Campbell, "Qinject: A virtual-machine based fault injection framework," in *13th Int. Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS '08)*, 2008, (Poster Presentation).

[6] S. Potyra, V. Sieh, and M. D. Cin, "Evaluating fault-tolerant system designs using FAUmachine," in *2nd Int. W'shop on Engin. Fault Tolerant Sys. (EFTS '07)*.

[7] F. M. David and R. H. Campbell, "Building a self-healing operating system," in *3rd IEEE Int. Symp. on Dep., Auton. & Secure Comp.* IEEE, 2007, pp. 3–10.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *11th Eur. Conf. on OOP (ECOOP '97)*, ser. LNCS, M. Aksit and S. Matsuoka, Eds., vol. 1241. Springer, Jun. 1997.

[9] O. Spinczyk, D. Lohmann, and M. Urban, "Advances in AOP with AspectC++," in *New Trends in Softw. Method., Tools & Techn. (SoMeT '05)*, ser. Frontiers in AI & Applications, no. 129. Tokyo, Japan: IOS Press, Sep. 2005, pp. 33–53.

[2]The *pointcut* expression, utilizing wildcard expressions (... and %), actually matches twelve different Bochs functions which implement memory writes for several address and data operand sizes.