FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance

Horst Schirmeier[†], Martin Hoffmann[‡], Christian Dietrich[‡], Michael Lenz[†], Daniel Lohmann[‡], and Olaf Spinczyk[†]
[†]Department of Computer Science 12
Technische Universität Dortmund, Germany
{horst.schirmeier,michael.lenz,olaf.spinczyk}@tu-dortmund.de
[‡]Chair of Distributed Systems and Operating Systems
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
{hoffmann,dietrich,lohmann}@cs.fau.de

***Abstract*—Due to voltage and structure shrinking, the influence of radiation on a circuit's operation increases, resulting in future hardware designs exhibiting much higher rates of soft errors. Software developers have to cope with these effects to ensure functional safety. However, software-based hardware fault tolerance is a holistic property that is tricky to achieve in practice, potentially impaired by every single design decision.**

We present FAIL*, an open and versatile architecture-level fault-injection (FI) framework for the continuous assessment and quantification of fault tolerance in an iterative software development process. FAIL* supplies the developer with reusable and composable FI campaigns, advanced pre- and post-processing analyses to easily identify sensitive spots in the software, well-abstracted back-end implementations for several hardware and simulator platforms, and scalability of FI campaigns by providing massive parallelization. We describe FAIL*, its application to the development process of safety-critical software, and the lessons learned from a real-world example.

I. Introduction

Chip technology is continuously moving towards higher densities and lower operating voltages [1] at the price of dramatically increasing sensitivity to electromagnetic radiation [2], [3], [4], [5]. Consequently, future hardware designs for embedded systems will exhibit an increasing rate of soft errors even on sea level. This trend creates new challenges for the development of reliable embedded systems, such as automotive control units. Certification authorities demand explicit measures to cope with transient faults in their functional safety standards (e.g., ISO 26262 [6]). However, for cost-sensitive mass products – such as cars – manufacturers cannot simply employ multiple redundant hardware components, as common in avionic systems. Instead, the problem has to be dealt with (at least partly) in the software: To not diminish all gains from these new hardware designs, embedded software developers have to *selectively* place application-specific error detection [7] and recovery mechanisms [8], [9] (EDM/ERMs) in their mixed-criticality systems. Critical tasks and sensitive spots in the software stack must be hardened against hardware faults, while the remaining less critical components economize resource consumption by occasionally tolerating incorrect results.

However, experience shows that this is difficult to achieve in practice: Against intuition, software-based EDM/ERMs often cause more harm than good [10], as their overhead in time and space also increases the "attack surface" of the system. Furthermore, even small changes to the functional part of the software, such as refactoring an array-based algorithm into one that uses pointer-based linked lists, can have a dramatic impact on the robustness. Hence, analogous to common practices in iterative software development, the soft-error analysis and hardening process must be iterated and supported by a *continuous fault-tolerance assessment* process. This process keeps developers informed about the robustness of their software, and repeatedly *quantifies* the effectiveness and efficiency of their hardening measures.

A. About this Paper

We present FAIL*[1], a flexible and versatile architecture-level fault injection (FI) tool and framework for developers designing and deploying hardware fault-tolerance measures. FAIL* is a result of several years worth of experience in FI, continuous fault-tolerance assessment, and quantitative comparison of programs' susceptibility to hardware faults. It has so far been used in at least 18 peer-reviewed publications conducted by more than 20 researchers from four different research groups, who have employed FAIL* to analyze the impact of robustness measures in a broad range of system software, including:

- General-purpose operating systems, such as Linux [11] and L4/Fiasco [12].
- Embedded real-time operating systems, including eCos [9], [13], [14], [15], CiAO [13], *d*OSEK [16], and ERIKA [16].
- The KESO JVM for embedded systems [17].
- The error-resilient CoRed voter [18], [19].
- Various benchmarks, such as the MiBench suite [14] or the nanojpeg decoder [20].

[1]The acronym FAIL* stands for FAult Injection Leveraged, with the asterisk highlighting its variability regarding target back ends.

Hence, we now consider it as ready to release[2] it under an open-source license to a wider audience.

Architecture-level FI has been the standard analysis technique in the software-based hardware fault-tolerance community for at least two decades [21], [22], [23], [24], [25]. It can be used for the localization of sensitive spots [9], delivering input for *fault removal* [26], and the quantitative evaluation of the effectiveness of fault-tolerance measures deployed in a specific system, also known as *fault forecasting* [26]. Note that FI is also used for the injection of *software faults* (i.e., programming errors); FAIL* may also be usable for this purpose, but the tooling has been focused on the injection of hardware faults.

Current FI tools are basically "expert tools". They are typically employed for very specific experiments, but hardly support a continuous and iterative process of measuring and improving software robustness from the very beginning. Many are exclusively intended for *testing* fault-tolerance measures but not for *quantitatively measuring* the whole system's resilience. Additionally they particularly lack (1) detailed post-injection analysis steps that guide the developer to converge towards optimally protected software; (2) support for many different target architectures and back ends, including the easy combination of test-port and simulation-based FI; (3) flexibility with respect to reusing and tailoring of FI campaigns; (4) the capability to easily scale FI coverage, the ratio of the total fault-space size and actually injected faults, with the available computing time and power.

B. Our Contributions

With FAIL*, we provide a unifying solution to these requirements under one roof. From the beginning [27], the tool was intended not only for testing, but also for measuring the resiliency of software systems to hardware faults. Based on state-of-the-art static analysis capabilities (involving the LLVM compiler infrastructure [28]), FAIL* comes with (1) a set of helper utilities automating post-injection analysis steps (including novel visualization capabilities), tuned for the vast amount of data to be expected from large-scale FI campaigns. Inspired by GOOFI's [29] flexible architecture, FAIL* offers (2) an abstraction layer for different simulator and hardware back ends. This abstraction layer is substantiated by (3) an experiment API with well-chosen primitives that allow reusing FI campaigns across back ends, complemented by experiment modules simplifying common tasks (such as tracing and output recording). With several different pre-injection (or *pruning*) techniques and parallelization capabilities, FAIL* allows to (4) scale the FI coverage tailored to the analysis needs and available computing power.

In combination with a full-system simulator (such as Bochs [30] or Gem5 [31]) or embedded development hardware (such as the PandaBoard ES [32]) this allows to measure and compare the error resiliency of complete software stacks, enabling, for example, research on dependable operating systems. Besides its utility for analyzing software systems, FAIL* constitutes a *meta experimentation* laboratory for FI techniques that can be used both in teaching and at the research frontier.

To summarize, the contributions of this article are:

- The description of a flexible and open FI tool for *both testing and quantitatively measuring* software-based fault-tolerance mechanisms deployed in software systems,
- with an architecture that allows *switching target back ends* with little effort, supported by an API abstracting away target back-end details and thereby fostering *experiment code reuse* (Section III),
- and advanced *pre- and post-injection analysis techniques* leveraging large-scale FI campaigns, helping at drawing conclusions from the results, and supporting a *continuous* fault-tolerance *assessment* process, including an experience report describing FAIL*'s usage during the development of a dependable real-time operating system (Section IV).

Section II reviews the state of the art in FI techniques and tools, Section V discusses some of FAIL*'s characteristics, and Section VI concludes the paper.

II. Fault-Injection Techniques and Tools

Since the identification of primary and secondary physical causes for soft errors in the 1970s [33], [34], several techniques for the artificial injection of hardware faults have been devised [21], [35], [36], [37], [24], [25]. Each technique tries to imitate the effects of naturally occurring causes (e.g., alpha or neutron radiation, power supply disturbances, or crosstalk) to a certain degree, while dramatically increasing the fault *probability* to a level usable for testing circuits and software. (Note that this paper does not address *software faults*, i.e., programming errors caused by humans).

Fault-injection techniques are characterized in their degree of *repeatability* (ability to inject a specific fault and obtain the same result), *controllability* (when and where to inject a fault), *intrusiveness* (impact on the target system), experiment result *observability* (ability to observe/measure the effects of an injection), and fault location *reachability* (how much of the CPU and periphery state is accessible for FI) [38].

A. Hardware-Implemented Fault Injection

Early on, *hardware-implemented* FI techniques were used, trying to closely imitate the natural sources of hardware faults. Gunneflo, Karlsson et al. [39], [40] expose CPUs and memory banks to heavy-ion radiation. Karlsson et al. [41], Miremadi and Torin [42], and Tummeltshammer and Steininger [43] use power-supply disturbances to provoke faults. Used as fault-injection techniques, heavy-ion radiation and power-supply disturbances have a low repeatability and controllability, a limited observability of the fault effects, and a high cost (particularly for radiation exposure) for the experiment setups in common [24], [25].

RIFLE [44] and MESSALINE [21] are examples of FI tools that inject faults through probes attached to the connector pins of CPU chips. Other tools, such as GOOFI-2 [38],

[2] FAIL* releases can be downloaded from: https://github.com/danceos/fail

Xception [45], and Fidalgo et al. [46], use test access ports (in these examples, Nexus [47] or JTAG [48]) to inject faults into the target system. Pin-level FI and especially FI via test access ports exhibit a better repeatability and controllability than their aforementioned hardware-implemented counterparts, but at the cost of injection speeds limited by the test access port, and limited reachability of injectable state.

All hardware-implemented FI techniques require specialized hardware setups, and only FI via test access ports can be achieved with COTS hardware while retaining the repeatability and controllability necessary for detailed post-injection analysis.

B. Software-Implemented Fault Injection

Much more cost-effective alternatives are several variants of *software-implemented* FI (SWIFI). In *pre-runtime* SWIFI, the target-system's software or data is injected with faults before it is run, as, for example, used by GOOFI [29] and Fuchs [49]. The primary drawbacks are the possibility of a "probe effect" [24] (high intrusiveness), limited reachability of injectable state, and a longer round-trip when the injection location is changed. The more widely used *runtime* SWIFI adds software to the target system that is triggered, for example, by exceptions or debugging features in CPUs, and injects faults into the running system. Examples are FIAT [50], FERRARI [51], Xception [45], and GOOFI-2 [38]. Eliminating the injection round-trip of pre-runtime SWIFI, the probe effect and limited reachability remains with runtime SWIFI.

C. Simulation-Based Fault Injection

Simulation-based FI [22], [25] injects faults into simulated hardware. Examples for tools that inject faults in low-level hardware models are VERIFY [52] and MAFALDA [53]. Relyzer [23] injects into a commercial functional full-system simulator on ISA level, and uses an additional microarchitectural and memory timing simulator in a short time frame around the injection. F-SEFI [54] and Qinject [55] inject faults into QEMU [56], a purely behavior-level machine simulator. The main drawback of simulation-based approaches is the reduced speed – especially low-level simulators tend to be many orders of magnitude slower than the real hardware. Nevertheless, simulation-based FI avoids the intrusiveness of SWIFI, offers high controllability and repeatability, and extends the reachability and observability to the detail level the simulator provides. Other advantages are capabilities like checkpointing, that open up optimization potential impossible to achieve with other FI approaches (e.g., using checkpoints to speed up experiments [57]), and the possibility to run FI experiments on any available hardware (e.g., on a computing cluster). The latter is especially advantageous in early product stages when no prototype hardware is available yet.

D. FAIL*: Fault-Injection Techniques and Unique Features

One of FAIL*'s driving design decisions was to allow developers to profit from its continuous fault-tolerance assessment capabilities (Section IV) without having to decide for a specific FI technique (and live with its advantages and disadvantages). Therefore, FAIL* currently supports *three* of the FI techniques mentioned in the previous sections that provide the level of repeatability and controllability necessary for detailed post-injection analysis: simulation-based FI – with currently three different simulator target back ends (Bochs [30], Gem5 [31], and QEMU [56], and a Synopsys CoMET back end currently under development) and two target architectures (x86-32 and ARM) – and a hybrid technique between test-port–based FI (injecting into JTAG-controlled ARM Cortex-A9 development boards, such as the PandaBoard [32]) and SWIFI (enhancing the observability by additional components on the target system).

Integrating these techniques in one tool allows the developer to switch back and forth from simulators to hardware during the development of a growing software system, and even to a different target architecture if the requirements change. Besides GOOFI/GOOFI-2 [29], [38], we know no FI tool with FAIL*'s back end flexibility while maintaining a uniform interface towards the experiment description.

Almost all FI tools only sample small parts of the fault space to achieve an estimate of the target's overall fault resiliency. Only recent works, such as Relyzer [23], or SmartInjector [58] can cover the whole (ISA-level) fault space of the target application by employing heuristic pruning methods, but lack post-injection analyses to fully profit from the obtained results. FAIL* as well has the capability to cover the whole fault space, but also offers detailed post-injection analyses down to the level of single variables, CPU instructions, or high-level program code lines (Section IV-D). FAIL* achieves this by advanced fault-space pruning techniques (Section IV-B), and massive parallelization (Section IV-C), allowing to unleash the power of computing clusters. Additionally, most FI tools in existence are specifically tailored for *testing*, but not for quantitatively comparing the hardware fault tolerance of specific applications protected by hardening techniques, which FAIL* explicitly supports [59].

In contrast to many FI tools that were never released to the public, such as GOOFI [29], or are only available commercially, such as Xception [45], FAIL* can be used for simulation-based FI out of the box[3] using open-source software only. Test-port–based FI requires only small investments in an ARM development board and a JTAG debugger.

III. FAIL* Architecture

At the topmost level, FAIL* is organized in a client/server architecture (Figure 1). In order to facilitate parallelization of FI experiments, the *Campaign Controller* hands out previously defined job parameters from a central database to FAIL* *instances* (e.g., running in a computing cluster). Conversely, experiment results are collected and stored in the database for subsequent analyses. This parallel execution requires that the experiment and its executive FAIL* instance behave *deterministic* regardless of the computing node it is hosted on.

A FAIL* *instance* is, in principle, a tapped existing simulator or debugger: FAIL* extends the back-end code with callback

[3] We provide a ready-to-run Docker-based demo campaign.

hooks at various crucial code locations. This allows to intercept and control the back-end execution and gain access to the (simulated) system state.

FAIL*'s key component, the Execution-Environment Abstraction (EEA), provides a common interface for the different target back ends. Its C++ API offers access to both target back-end meta-information and the current state, and allows registering *Listeners* for several types of events. The FAIL* API currently provides abstractions for:

- **Meta-information** on the target back end: Number of CPUs, number of registers, platform-independent naming of special registers (program counter, stack pointer), bit widths and byte order, memory size.
- **Fine and coarse-grained state** access: Read/write access to CPU registers and memory, injection of external interrupts, access to the back end's time; save/restore of the back-end state, and reboot.
- Listeners registerable for **events in the back end**: Reaching specific program instructions (similar to a breakpoint), access to specific memory addresses, CPU exceptions, external interrupts, serial I/O, passing of specific amounts of back-end time.

Each target back end may additionally introduce interfaces to target-specific state, such as a means to manipulate a network device. Naturally, experiments using such an interface cease being portable to a different target, unless an adequate abstraction is added to the generic API.

The actual (user-defined) experiment controls the attached back end through the EEA. The next section will provide more details on experiment definition, and fault models that can be implemented thereby.

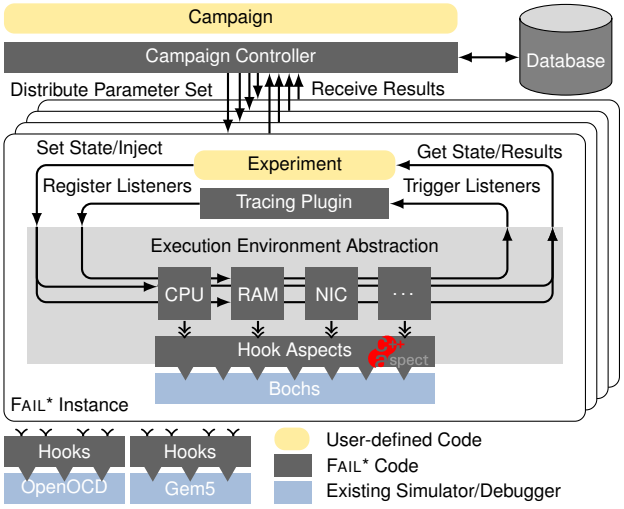

Fig. 1. FAIL* architecture overview: The *Campaign Controller* distributes parameter sets from a user-defined *Campaign* throughout the FAIL* instances. Each FI experiment consumes a parameter set, and controls its target back end through the *Execution-Environment Abstraction* (EEA) layer. Actual target back ends (simulators, or real prototype hardware) can be exchanged by providing an interfacing module to this abstraction.

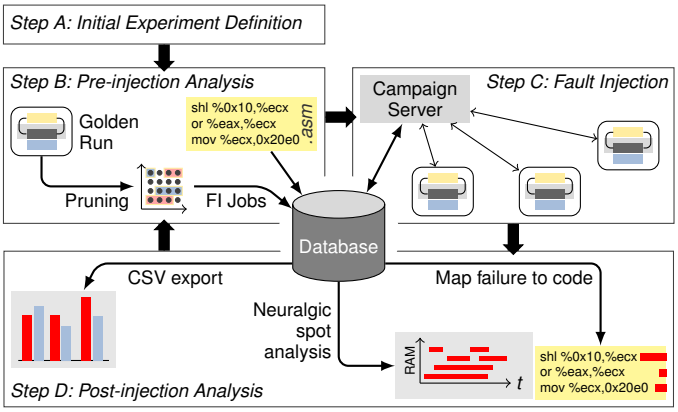

Fig. 2. The fault-tolerance assessment cycle: After an initial *experiment definition* step, the developer enters one or multiple iterations of the fault-tolerance assessment cycle, improving the system's fault resilience in each iteration (Steps A to D correspond to Sections IV-A to IV-D).

IV. The Fault-Tolerance Assessment Cycle

In this section, we explain the steps involved in the fault-tolerance assessment cycle (Figure 2) that allows the developer to converge to an optimally protected software stack, and how FAIL* assists each of these steps. The steps are supported by and applied to a running example, the fault-tolerance assessment of *d*OSEK, an embedded real-time operating system [16] with fault tolerance as a first-class design goal.

A central goal of *d*OSEK is to provide functional safety even in the presence of transient hardware faults. Thus, the operating system not only has to execute the application tasks in accordance with the specified real-time behavior, but also has to detect any transient fault during the execution of the kernel. To cover both aspects, *d*OSEK's automated build process invokes fine-grained unit tests to uncover software faults in specific components of the OS (e.g., scheduler, dispatcher, interrupt handling) in combination with FI experiments continuously evaluating the robustness of the system against hardware faults with FAIL*.

The goal of continuous FI is to uncover critical spots of the system, and to evaluate the effectiveness of additionally applied fault-tolerance measures in an iterative process. The robustness evolution, as shown in Figure 3, can be logged in the version control system, which allows to evaluate the effectiveness of different measures – or adverse effects of implementation flaws.

FAIL* supports the developer in all steps required for extensive FI campaigns aiding fault removal or fault forecasting. The following Sections IV-A through IV-D describe *experiment definition*, *pre-injection analysis*, the *FI campaign*, and the *post-injection analysis*, which also correspond to steps A–D in Figure 2. In Section IV-E, we give some details on FAIL*'s implementation, and platform requirements.

A. Experiment Definition

In the initial experiment definition step (Figure 2, step A), the developer defines an experiment procedure to be fed with experiment parameters by the Campaign Controller

(cf. Figure 1), and to be run in every single FI experiment. Essentially, the developer thereby decides which parameters are fixed during the FI campaign, and which are variable – or, in other words, chooses the fault model and spans the experiment-parameter space. Later, step C – the FI campaign – will walk this parameter space and conduct actual experiments.

Defining the experiment procedure can be accomplished by picking generic code templates that, for example, flip a single bit in the register file at a specified point in time, and observe a standard set of output behaviors of the target software. In this case, the parameter space is spanned by the time (measured in CPU cycles from the program start) and location (register name and bit number) for the fault injection. The corresponding experiment procedure essentially implements one injection run taking a coordinate from this parameter space as its input.

For a more complex fault model or special requirements on target behavior observation, the developer can extend a generic experiment and fill in the required, special behavior, or even write a complete experiment description from scratch. For example, the developer could easily implement an experiment in which integer division operations in the ALU yield faulty results, or every read access to odd memory addresses reads from the wrong memory cell – FAIL*'s C++ API enables a wide variety of use cases.

This degree of freedom allows the developer to experiment with different fault models if necessary, for example, single- or multi-bit, transient, intermittent, or permanent faults (the latter by re-writing a faulty value every time it changes), at different places in the memory hierarchy, or when specific types of instructions are executed. Table I lists examples for fault models that have already been implemented using FAIL*, the necessary preparation steps before running the campaign (cf. Sections IV-B and IV-C), and the EEA layer primitives (see also Section III) used in the experiment implementation.

In the normal case, when no particularly fancy fault models are needed, one of the previously mentioned generic code templates can be used instead. Optionally, the experiment can enable plugins (which are implemented using the same API as the experiments) that encapsulate often needed functionality, such as recording a memory-access trace.

Application to *d*OSEK: During *d*OSEK's development, single-bit flips in memory, general purpose registers, as well as the instruction pointer and flags register were selected as the fault model. The actual FI experiment is based on a standard FAIL* experiment: Fault parameters are retrieved from the Campaign Controller and injected accordingly. Additionally, the experiment was extended to evaluate specific operating-system functionality.

Listing 1 shows a simplified excerpt from the *d*OSEK injection experiment. To check the correct system behavior, the tasks of the system under test write a magic value to a global memory address (`g_trace_var`, line 2). In doing so, the FAIL* experiment code can record the task activation order and time by continuously observing the according memory location with the help of FAIL*'s *Checkpoint* plugin (line 17) – a ready-to-use plugin that traces the written value together with the current simulation time. Using the *ELFReader* helper class, the memory address of the global object is derived directly from the image file under test and used to install an appropriate *MemoryWriteListener* within the Checkpoint object.

For the actual injection of the bit flip, the system is executed to the FI time as defined in the experiment parameter set (lines 23, 27). Using the *MemoryManager*, FAIL* reads, manipulates, and updates the fault location (line 30).

Using the ElfReader, the instruction addresses of different experiment end points can be determined and used to set up appropriate listeners. Such end points are trap and breakpoint listeners. A dedicated shutdown function labels the intended end of a test run (line 41). When reaching this shutdown function, the traced task activation is evaluated (line 49); any deviation from the expected sequence (traced during the golden run) denotes the run as silent data corruption (SDC). The experiment further adds a *TimeoutListener* to cancel an irresponsive run after a certain amount of time. Finally, the experiment sends the outcome back to the Campaign Controller (lines 50, 52, 55, and 57), which logs all results in a database.

B. Pre-Injection Analysis

After the one-time preparation of a single FI experiment, the first step that gets re-visited in each iteration of the fault-tolerance assessment cycle is the *pre-injection analysis* (Figure 2, step B).

The main prerequisite for this analysis is a compiled binary image of the target-system software (usually an ELF image or bootable disk image) the chosen execution back end is able to run.[4] This image is then executed by a FAIL* instance with the previously defined experiment in "preparation mode": In this so-called *golden* run, no faults are injected. Instead, the normal behavior of the system (especially what the developer defined as its output channels, e.g., the serial interface) and the benchmark's run time is recorded. Additionally, an instruction and memory-access trace is saved for an important pre-injection step: static analysis and fault-space pruning.

Based on the recorded trace, FAIL* massively reduces the number of necessary FI experiments using a classic fault-space pruning method known as *operational profile-based pruning* [61] or *def/use pruning* [23]. This pruning method makes use of the fact that for any fault injected prior to a an instruction *writing* to the fault location, the fault gets masked. A fault injected prior to an instruction *reading* the fault location has the same effect regardless of its exact position in time before this instruction. Thus, write-read equivalence classes can be formed for each fault location, and used for FI. This pruning technique (implemented in the *import-trace* program) reduces the number of experiments needed to achieve 100 percent coverage of all possible fault-space coordinates (time-location pairs) while maintaining precision in identifying critical spots in the software.

[4]Usually, *multiple* different benchmarks combined with different software-based fault-tolerance configurations are assessed at once. For the sake of simplicity, we will continue to describe the process for a single image.

TABLE I
FAULT MODEL EXAMPLES IMPLEMENTED USING FAIL*, NECESSARY CAMPAIGN PREPARATION STEPS, AND EEA LAYER PRIMITIVES USED IN THE RESPECTIVE EXPERIMENT IMPLEMENTATION.

Fault Model	Preparation	Experiment Implementation	Publications
Transient *single*-bit flips in *data* memory, uniform	Golden-run memory-access trace analysis (using *tracing* plugin and *import-trace* tool)	*Fault trigger:* N CPU cycles passed; *Fault injection:* Memory or CPU abstraction in EEA layer	e.g., [9], [13], [15], [16], [59], [60]
Transient *single*-bit flips in *data* memory, on access *("inject-on-read")*			[59]
Transient *multi*-bit flips in *data* memory, uniform			e.g., [9], [14]
Transient *single*-bit flips in *instruction* op-codes	Golden-run instruction trace analysis (*tracing* plugin & *import-trace* tool)		[18], [19]
Transient *single-bit* flips in *register file*, uniform	Golden-run instruction trace analysis & disassembly (*tracing* plugin & *import-trace* tool)		[16], [60]
Transient *single*-bit flips in *register file*, on access *("inject-on-read")*			[18], [19]
Transient *multi*-bit flips in *register file*, on access *("inject-on-read")*			
Permanent single-bit flips in memory	—	*Fault trigger:* Memory write at faulty location; *Fault injection:* Memory abstraction in EEA layer (re-write faulty bit)	[11]

Depending on the fault model, an analysis of the static instructions executed during the golden run may be required as an additional input for the pruning technique. For larger benchmarks, an alternative heuristical pruning method (described in detail in [14]) can be used to reduce the experiment efforts even further. Alternatively, if detailed result analysis is not needed, classical fault-space sampling can be used.

Summing up, the pre-injection analysis distills the minimal necessary injections that still cover the predefined fault space, and yields a list of FI job parameters. The information recorded during the golden run additionally serves as a reference for what is deemed to be the correct program output and behavior.

Application to *d*OSEK: With the help of FAIL*'s def/use pruning techniques on memory and register accesses, the number of experiments needed to cover the complete ISA-level single-bit flip fault space for four different *d*OSEK variants was reduced from 4.8×10^{11} to about 32 million. With sampling, it can be reduced to tens of thousands, at the cost of removing the possibility for detailed analyses.

For the sake of reproducibility and to ease the post-injection analysis, the assembler code of the *d*OSEK system under test was added to the database together with a commit ID pointing to the according version of the software sources.

C. Fault-Injection Campaign

In the fully automated FI campaign (Figure 2, step C), the Campaign Controller distributes job parameters from the database to FAIL* client instances running on the same machine, or on potentially hundreds of computing cluster nodes. The results are collected in a database table automatically tailored specifically for the running campaign, which may, for example, need additional columns for recording the fault-detection latency or other experiment-specific values. For the subsequent post-injection analysis step, the Campaign Controller ensures that each result can be traced back to the exact state bit that was modified in the FI, and the point in time (down to the dynamically executed CPU instruction) the injection took place.

Application to *d*OSEK: FAIL*'s parallelization capabilities allowed to keep the campaign duration and thus the entire development cycle to a bearable level. With 100 FAIL* clients running (each occupying one host-CPU core), this campaign took about 36 hours to complete. With our computing cluster (≈3,000 CPUs), this runtime could be further reduced to under two hours, as single experiments can be conducted completely independent of each other.

D. Post-Injection Analysis

After collecting all FI results, FAIL*'s post-injection analysis features help the developer by quantifying the software fault-tolerance effectiveness (fault forecasting), or by aiding the process of system hardening (fault removal) and the placement of error detection (or recovery) measures (Figure 2, step D).

For quantifying the effectiveness of fault-tolerance measures, a reduced-effort sampling of the fault space (cf. Section IV-B) suffices, but naturally increases in preciseness with larger samples. A set of analysis scripts aggregate the FI results, if necessary extrapolate from the sample to the population size, and generate CSV tables that can be, for example, processed in GNU R to generate bar plots, as shown in Figure 3. Depending on the desired fault model, results can be pre-filtered (e.g., to include only results related to specific parts of the system-under-test) or weighted with the lifetimes of the values modified by the injection [61].

If the developer needs data for hardening the software system, such as for placing error-detection (or recovery) mechanisms on specific data structures, or on the output of critical program modules, a more fine-grained FI result analysis is necessary. The fault-space plot in Figure 4 shows a two-dimensional projection of the raw result data this information can be derived from: The X axis represents the time, discretized in CPU cycles

```
// Get trace symbol
ElfSymbol &s_trace = elf.get("g_trace_var");

while(campaign.hasJobs()) {
// Retrieve experiment parameter
dOSEKExperimentParam param =
 campaign.getParam();

// Check if injection would hit trace array
if (s_trace.addr == param.injection_addr) {
    m_log << "skip trace variable" << endl;
    campaign.sendReply("No Injection");
    continue; // with next parameter set
}

// Enable Checkpoint Plugin
// Compares results with previous Golden Run
Checkpoint cpt(s_trace, "golden_run.trace")
simulator.addPlugin(cpt);

// Prepare Breakpoint Listener
BPListener l_inject(ANY_ADDR);
l_inject.setCounter(param.step_count);
simulator.addListener(l_inject);

// Proceed to injection
// (after param.step_count instructions)
simulator.resume();

// inject transient single-bit fault
MemoryManager& m=simulator.getMemoryManager();
char value = m.getByte(param.injection_addr);
injectedval = value ^ (1 << param.bit_to_flip);
m.setByte(param.injection_addr, injectedval);

// Prepare result listeners
ElfSymbol &s_sd = elf.get("shutdown");
BPListener l_shutdown(s_sd);
TrapListener l_trap();
TimeoutListener l_timeout(10000); // 10 ms

simulator.add(l_shutdown);
simulator.add(l_trap);
simulator.add(l_timeout);
// Continue simulator
BaseListener *l = simulator.resume();

// Examine outcome
if (l == &l_shutdown) {
    if(cpt.valid()) {
        campaign.sendReply("OK");
    } else {
        campaign.sendReply("SDC");
    }
} else if (l == &l_trap) {
    campaign.sendReply("Trap");
} else if (l == &l_timeout) {
    campaign.sendReply("Timeout");
}
} // end of while
```

Listing 1. Simplified *d*OSEK FAIL* experiment: The code excerpt shows the FI part (using the FAIL* API via the global `simulator` object), parametrized by the injection address and the bit to flip. Among other details, this information was communicated to the experiment by the Campaign Controller.

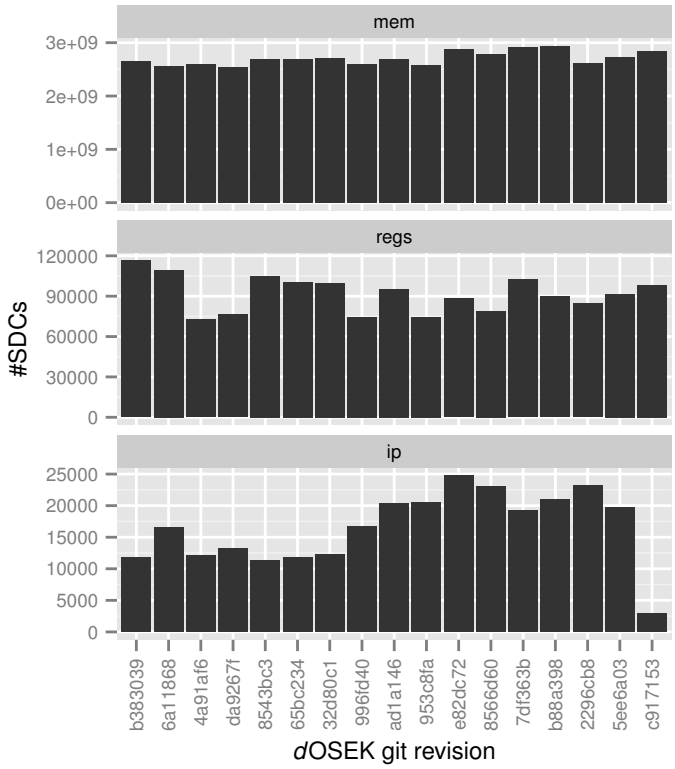

Fig. 3. Excerpt of the robustness development history of *d*OSEK . The influence on the robustness of each software change can be easily reproduced and analyzed.

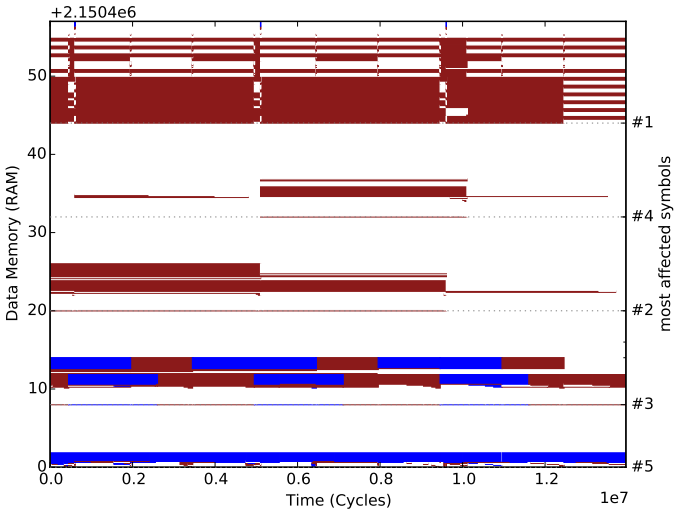

Fig. 4. Excerpt from a *d*OSEK fault-space plot, including references to the most SDC-sensitive symbols (Table IIa) at their corresponding start address, shown on the right margin. (Plot coloring: *red*: SDC, *blue*: Timeout, *white*: No effect).

since benchmark start; the Y axis denotes the state a fault was injected in, in this case the addresses of bytes in main memory (annotated with references to specific global data structures from Table IIa at their corresponding start address, shown on the right-hand side). The coloring of each coordinate (time-location pair) signifies the outcome of a single FI experiment in which the fault was injected at the specified point in time and address: *White* means the fault had *no effect*, while non-white colors denote different types of failure (*blue*: Timeout, *red*: Silent Data Corruption). This data can be collected by running FI experiments for *every* coordinate in this fault space, and applying a fault-space pruning technique (cf. Section IV-B) to reduce the number of experiments to a feasible amount. Note that the diagram does not plot the *detection latency*, that is, the time between the injection of a fault and its detection by software or the experiment.

Based on this raw data, FAIL* is capable of running a variety of analyses. One example is the "top N most critical data structures" list: By aggregating failure counts over time, and over data structures on the "state" axis, the latter can be ordered by their importance for the program's successful termination. The most susceptible data structures are a logical target for hardening measures, such as the protection with checksums. Similarly, failures can also be aggregated over the state axis, and over the dynamic execution of functions or modules in the program (Table IIb), yielding possible targets for function hardening, for example, with N-modular redundant execution.

Beyond these aggregation techniques, FAIL* is capable to map failures to static CPU instructions in the program, and from there to high-level language code using DWARF debugging information [62]. A browser-based visualization tool named *Visual*FAIL* helps the developer to pinpoint propagation leaks in the error detection code and eliminate remaining points of failure. Another recent, still experimental FAIL* feature based on debugging information, is the assignment of failures to short-lived data structures on the stack.

Application to *d*OSEK: While Figure 3 shows the failure count totals for several versions of *d*OSEK's evolution (which could also be achieved with low-effort sampling techniques on a single machine), FI results for the complete fault space were distilled for much more detailed analyses. Using the aforementioned post-processing steps, the final results of each of *d*OSEK's FI campaigns were arranged in an automated fashion to present the most vulnerable spots of the system in terms of injection time and location. Table II shows a "Top-5" analysis of a benchmark run of an unhardened *d*OSEK system in terms of SDC-sensitive memory locations. The fault-space plot, as shown in Figure 4, helped to match the silent data corruptions to specific points in time. After identifying the most vulnerable locations, VisualFAIL* (Figure 5) supported the developers to immediately inspect the critical spots on source code level and elaborate on the further hardening of the system.

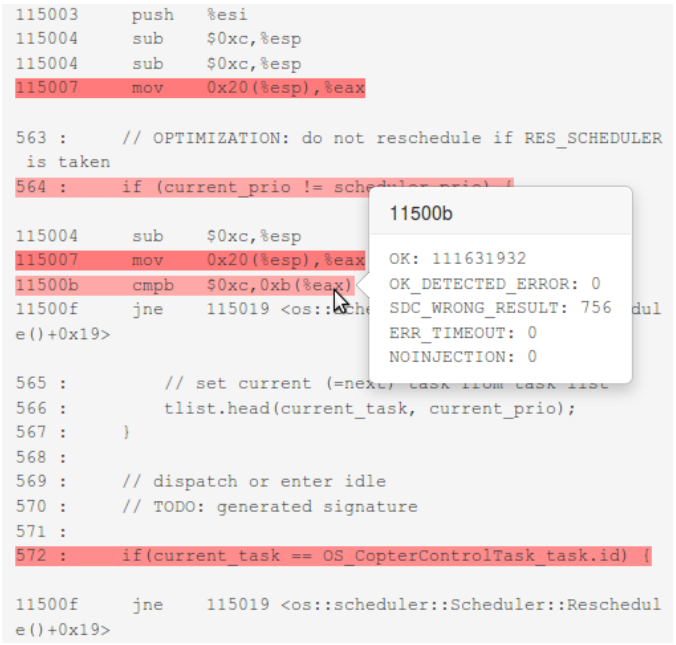

Fig. 5. Post-injection analysis: VisualFAIL* highlights source-code lines (interspersed with associated assembler instructions) activating faults that lead to SDCs.

E. Implementation and Platform Requirements

FAIL* is implemented in about 34,000 lines of C++ code, accompanied by a set of bash, Perl and Python scripts, and is shipped with the slightly modified source code of the currently supported execution back ends (Bochs [30], gem5 [31], QEMU [56], OpenOCD [63]). In order to ease the transition to newer simulator versions, and to properly separate the concerns *simulation* and *fault injection*, we use Aspect-Oriented Programming (in our case AspectC++ [64], an AOP extension to C++) to "hook" FAIL* into Bochs. All pre- and post-injection analysis steps involving static analysis steps rely on LLVM [28], and the data management is delegated to MySQL (or, preferredly, MariaDB). Most data visualization and plotting is handled by GNU R and Python's matplotlib.

FAIL* currently only runs on Linux (x86-32 and amd64 are known to work), although porting to other POSIX-compliant platforms should be possible with relatively low effort.

V. DISCUSSION

Revisiting the characteristics of fault-injection techniques [38], FAIL* can be classified as follows.

Repeatability: The simulation-based back ends of FAIL* ensure a deterministic execution behavior and therefore reliably reproducible experiments. All input sets and their according results, as well as the system-under-test image itself, are organized in a database, thus each specific experiment can be reconsidered and repeated easily. If FAIL* is integrated in an automated build process, even the entire robustness evolution can be revisited – revision by revision. The same repeatability can be achieved even for test-port–based FAIL* back ends, if

TABLE II
POST-INJECTION ANALYSIS OF A BENCHMARK RUN OF AN UNHARDENED *dOSEK* VARIANT. FAIL*'S AUTOMATED POST-PROCESSING OF RESULTS ALLOWS THE DEVELOPER TO IMMEDIATELY IDENTIFY THE MOST VULNERABLE SPOTS OF THE SYSTEM.

#	Symbol	Address	Size	SDC	(%)
1	os::scheduler::scheduler_	0x20d02c	13	888,954,297	(62.9 %)
2	os::OS_FCctrl_alarm	0x20d014	12	223,003,587	(15.8 %)
3	os::OS_SG_alarm	0x20d008	12	196,283,827	(13.9 %)
4	os::OS_CCtrlWDAlarm_alarm	0x20d020	12	83,387,077	(5.9 %)
5	_sdata_os	0x20d000	8	13,059,577	(0.9 %)

(a) Top five SDC-sensitive symbols (or, contiguous memory areas).

Source File	SDC count	(%)
dosek.cc	1,100,408,145	(78.3 %)
os/alarm.h	291,248,981	(20.7 %)
os/counter.h	13,053,352	(0.9 %)
arch/i386/syscall.cc	39,447	(< 0.1 %)
arch/i386/dispatch.cc	25,738	(< 0.1 %)

(b) Source-code files where the faults for most SDCs get activated.

any external inputs, mainly periodic or sporadic interrupts, are under the control of the experiment.

Intrusiveness: Simulation-based FI solutions do not affect the original source code of the system-under-test. Therefore, the only possible "probe effect" can origin from an influence on the real-time behavior of the system. Using FAIL* with a simulation-based back end avoids this problem, as the entire system, including any peripherals, are under control, and can be suspended at any point in time. The intrusiveness of the hybrid test-port/SWIFI back end can be minimized but not completely avoided, for example if the JTAG interface ensures that the the CPU and timer subsystems are fully stopped while handling an FI event. Nevertheless, peripherals that are not under the control of the debugging system may still lead to undesirable interference.

Reachability: Here, FAIL* is clearly depending on the verboseness of the controlled back-end system. Regarding the simulator-based back ends, FAIL* observes and provides the entire system state – as far as actually simulated – through the Execution-Environment Abstraction (EEA). Test-port–based back ends, on the other hand, are restricted by the capabilities of the debugging hardware attached to the prototyping board.

Controllability: FAIL*'s memory and register managers provide full control over the state of the system-under-test via the EEA. FAIL* also provides means to save the entire system state, for example right before the first injection time during a golden run. During the FI campaign, each experiment can then bypass lengthy bootup/startup times by just restoring the system state and proceeding to the injection time. Further, arbitrary external interrupts, as well as a full system reboot can be triggered from the running experiment.

Observability: This aspect is closely connected with the reachability and controllability of the FI back end. FAIL*'s listener concept allows to observe and react on various kinds of events during an experiment run. While the test-port–based back ends are generally restricted to events that can be mapped to watch- or breakpoints and exception handlers in the additional SWIFI components, a simulation-based back end provides more elaborate listener activations. Here, the simulation-based variants allow for an unbounded number of listeners, while a hardware back end is often restricted, for example by a limited number of hardware breakpoints.

Besides the aforementioned functional aspects, an important design goal of FAIL* is to provide developers a convenient way to assess the fault tolerance of their systems. FAIL* already provides different ready-to-use experiments that can be easily extended to specific needs using the target-independent EEA. FAIL*'s client/server architecture facilitates massive parallelization of FI experiments, which allows full fault-space coverage even for more complex campaigns. Finally, using VisualFAIL* the developer can immediately inspect the FI results down to assembler-level instructions and high-level source code, and work out specific fault-tolerance measures.

VI. SUMMARY

Fault injection is a key element in the development process of safety-critical systems. FAIL* aims to provide a flexible tool set for even large-scale fault-injection campaigns with a target-independent execution-environment abstraction. High scalability in combination with different fault-space pruning techniques allow to scale the available computing time and power, up to full fault-space coverage. Supported by detailed post-injection analyses that immediately point out critical spots and assist in fault-removal design strategies, FAIL* encourages *continuous fault-tolerance assessment*, as we demonstrated on the *d*OSEK development example.

Apart from its practical purpose as a fault-injection tool set for both teaching and research, FAIL* can equally be seen as a foundation for further research and development on fault-injection and fault-space pruning concepts itself. During the last few years, FAIL* emerged to a versatile tool set that was already involved in at least 18 publications by 20 researchers from four different research groups. We now release FAIL* to a wider audience under an open-source license, and look forward to a community-driven evolution of FAIL* – and its application in further projects.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their comments. We also thank all additional contributors to FAIL*, especially Adrian Böckenkamp, Christoph Borchert, Björn Döbel, Tobias Friemel, Richard Hellwig, Florian Lukas, and Lars Rademacher. Kudos also go to Björn Bönninghoff for suggestions improving the readability of this paper.

This work was partly supported by the German Research Foundation (DFG) priority program SPP 1500 under grants no. SP 968/5-3 and LO 1719/1-3.

REFERENCES

[1] International Roadmap Committee, "International technology roadmap for semiconductors, 2013 edn. (executive summary)," *Semiconductor Industry Association*, 2013.

[2] S. Y. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[3] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 258–266, May 2005.

[4] V. Narayanan and Y. Xie, "Reliability concerns in embedded system designs," *IEEE Computer*, vol. 39, no. 1, pp. 118–120, 2006.

[5] M. Duranton, S. Yehia, B. de Sutter, K. de Bosschere, A. Cohen, B. Falsafi, G. Gaydadjiev, M. Katevenis, J. Maebe, H. Munk, N. Navarro, A. Ramirez, O. Temam, and M. Valero, "The HiPEAC vision," Network of Excellence on High Performance and Embedded Architecture and Compilation, Tech. Rep., 2010.

[6] ISO, *ISO 26262-6:2011: Road vehicles – Functional safety – Part 6: Product development at the software level*. Geneva, Switzerland: International Organization for Standardization, 2011.

[7] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *Proceedings of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '12)*. IEEE Computer Society Press, Jun. 2012, pp. 1–12.

[8] K. Pattabiraman, V. Grover, and B. G. Zorn, "Samurai: Protecting critical data in unsafe languages," in *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys '08)*. New York, NY, USA: ACM Press, 2008, pp. 219–232.

[9] C. Borchert, H. Schirmeier, and O. Spinczyk, "Generative software-based memory error detection and correction for operating system data structures," in *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*. IEEE Computer Society Press, Jun. 2013.

[10] A. Martínez-Álvarez, S. A. Cuenca-Asensi, F. Restrepo-Calle, F. R. P. Pinto, H. Guzmán-Miranda, and M. A. Aguirre, "Compiler-directed soft error mitigation for embedded systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 2, pp. 159–172, Mar. 2012.

[11] H. Schirmeier, I. Korb, O. Spinczyk, and M. Engel, "Efficient online memory error assessment and circumvention for Linux with RAMpage," *International Journal of Critical Computer-Based Systems*, vol. 4, no. 3, pp. 227–247, 2013, special Issue on PRDC 2011 Dependable Architecture and Analysis.

[12] B. Döbel, "Operating system support for redundant multithreading," Dissertation, TU Dresden, 2014.

[13] M. Hoffmann, C. Borchert, C. Dietrich, H. Schirmeier, R. Kapitza, O. Spinczyk, and D. Lohmann, "Effectiveness of fault detection mechanisms in static and dynamic operating system designs," in *Proceedings of the 17th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '14)*. IEEE Computer Society Press, Jun. 2014, pp. 230–237.

[14] H. Schirmeier, C. Borchert, and O. Spinczyk, "Rapid fault-space exploration by evolutionary pruning," in *Proceedings of the 33rd International Conference on Computer Safety, Reliability and Security (SAFECOMP '14)*, ser. Lecture Notes in Computer Science. Springer-Verlag, Sep. 2014, pp. 17–32.

[15] C. Borchert, H. Schirmeier, and O. Spinczyk, "Generic soft-error detection and correction for concurrent data structures," *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, 2015, to appear.

[16] M. Hoffmann, F. Lukas, C. Dietrich, and D. Lohmann, "dOSEK: The design and implementation of a dependability-oriented static embedded kernel," in *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications (RTAS '15)*. Los Alamitos, CA, USA: IEEE Computer Society Press, Apr. 2015.

[17] I. Stilkerich, M. Strotz, C. Erhardt, M. Hoffmann, D. Lohmann, F. Scheler, and W. Schröder-Preikschat, "A JVM for soft-error-prone embedded systems," in *Proceedings of the 2013 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '13)*, Jun. 2013, pp. 21–32.

[18] M. Hoffmann, P. Ulbrich, C. Dietrich, H. Schirmeier, D. Lohmann, and W. Schröder-Preikschat, "A practitioner's guide to software-based soft-error mitigation using AN-codes," in *Proceedings of the 15th IEEE International Symposium on High Assurance Systems Engineering (HASE '14)*. Miami, Florida, USA: IEEE Computer Society Press, Jan. 2014, pp. 33–40.

[19] ——, "Experiences with software-based soft-error mitigation using AN codes," *Software Quality Journal*, pp. 1–27, Nov. 2014.

[20] B. Döbel, H. Schirmeier, and M. Engel, "Investigating the limitations of PVF for realistic program vulnerability assessment," in *Proceedings of the 5rd HiPEAC Workshop on Design for Reliability (DFR '13)*, Berlin, Germany, Jan. 2013.

[21] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166–182, Feb. 1990.

[22] A. Benso and P. E. Prinetto, *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, ser. Frontiers in electronic testing. Boston, Dordrecht, London: Kluwer Academic Publishers, 2003.

[23] S. K. Sastry Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*. New York, NY, USA: ACM Press, 2012, pp. 123–134.

[24] 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, pp. 171–186, 2004.

[25] M. Kooli and G. Di Natale, "A survey on simulation-based fault injection tools for complex systems," in *Proceedings of the 9th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS '14)*. IEEE Computer Society Press, May 2014, pp. 1–6.

[26] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[27] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk, "FAIL*: Towards a versatile fault-injection experiment framework," in *25th International Conference on Architecture of Computing Systems (ARCS '12), Workshop Proceedings*, ser. Lecture Notes in Informatics, G. Mühl, J. Richling, and A. Herkersdorf, Eds., vol. 200. German Society of Informatics, Mar. 2012, pp. 201–210.

[28] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Los Alamitos, CA, USA: IEEE Computer Society Press, Mar. 2004.

[29] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: Generic object-oriented fault injection tool," in *Proceedings of the 31st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '01)*. Los Alamitos, CA, USA: IEEE Computer Society Press, Jun./Jul. 2001, pp. 83–88.

[30] K. P. Lawton, "Bochs: A portable PC emulator for Unix/X," *Linux Journal*, vol. 1996, no. 29es, p. 7, 1996.

[31] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[32] "Pandaboard homepage," http://pandaboard.org.

[33] D. Binder, E. Smith, and A. Holman, "Satellite anomalies from galactic cosmic rays," *IEEE Transactions on Nuclear Science*, vol. 22, no. 6, pp. 2675–2680, Dec. 1975.

[34] T. C. May and M. H. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Transactions on Electron Devices*, vol. 26, no. 1, pp. 2–9, Jan. 1979.

[35] J. A. Clark and D. K. Pradhan, "Fault injection: A method for validating computer-system dependability," *IEEE Computer*, vol. 28, no. 6, pp. 47–56, Jun. 1995.

[36] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, Apr. 1997.

[37] J. V. Carreira, D. Costa, and J. G. Silva, "Fault injection spot-checks computer system dependability," *IEEE Spectrum*, vol. 36, no. 8, pp. 50–55, Aug. 1999.

[38] D. Skarin, R. Barbosa, and J. Karlsson, "GOOFI-2: A tool for experimental dependability assessment," in *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '10)*. Los Alamitos, CA, USA: IEEE Computer Society Press, Jun./Jul. 2010, pp. 557–562.

[39] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation," in *Proceedings of the 19th Annual International Symposium on Fault-Tolerant Computing (FTCS '89)*. IEEE Computer Society Press, Jun. 1989, pp. 340–347.

[40] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo, "Using heavy-ion radiation to validate fault-handling mechanisms," *IEEE Micro*, vol. 14, no. 1, pp. 8–23, Feb. 1994.

[41] J. Karlsson, U. Gunneflo, P. Lidén, and J. Torin, "Two fault injection techniques for test of fault handling mechanisms," in *Proceedings of the 1991 International Test Conference (ITC '91)*, Oct. 1991.

[42] G. Miremadi and J. Torin, "Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection," *IEEE Transactions on Reliability*, vol. 44, no. 3, pp. 441–454, Sep. 1995.

[43] P. Tummeltshammer and A. Steininger, "Power supply induced common cause faults – experimental assessment of potential countermeasures," in *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '09)*. IEEE Computer Society Press, Jun./Jul. 2009, pp. 449–457.

[44] H. S. Madeira, M. Rela, F. Moreira, and J. G. Silva, "RIFLE: A general purpose pin-level fault injector," in *Proceedings of the 1st European Dependable Computing Conference (EDCC '94)*, K. Echtle, D. Hammer, and D. Powell, Eds. Springer-Verlag, 1994, pp. 197–216.

[45] J. Carreira, H. S. Madeira, and J. G. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 125–136, Feb. 1998.

[46] 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 Press, 2006, pp. 214–219.

[47] IEEE-ISTO, *The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface*. IEEE Computer Society Press, 1999.

[48] C. M. Maunder and R. E. Tulloss, *The Test Access Port and Boundary-Scan Architecture*. IEEE Computer Society Press, 1990.

[49] E. Fuchs, "An evaluation of the error detection mechanisms in MARS using software-implemented fault injection," in *Proceedings of the 2nd European Dependable Computing Conference (EDCC '96)*, A. Hlawiczka, J. G. Silva, and L. Simoncini, Eds. Springer-Verlag, 1996, pp. 73–90.

[50] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, "Fault injection experiments using FIAT," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 575–582, Apr. 1990.

[51] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A flexible software-based fault and error injection system," *IEEE Transactions on Computers*, vol. 44, pp. 248–260, 1995.

[52] V. Sieh, O. Tschäche, and F. Balbach, "VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions," in *Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing (FTCS '97)*, Jun. 1997, pp. 32–36.

[53] J. Arlat, J.-C. Fabre, M. Rodríguez, and F. Salles, "Dependability of COTS microkernel-based systems," *IEEE Transactions on Computers*, vol. 51, pp. 138–163, 2002.

[54] Q. Guan, N. Debardeleben, S. Blanchard, and S. Fu, "F-SEFI: A fine-grained soft error fault injection tool for profiling application vulnerability," in *Proceedings of the 28th IEEE Parallel and Distributed Processing Symposium (PDPS '14)*. IEEE Computer Society Press, May 2014, pp. 1245–1254.

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

[56] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005, pp. 41–46.

[57] L. Berrojo, I. Gonzalez, F. Corno, M. Reorda, G. Squillero, L. Entrena, and C. Lopez, "New techniques for speeding-up fault-injection campaigns," in *Proceedings of the 2002 Conference on Design, Automation & Test in Europe (DATE '02)*. IEEE Computer Society Press, 2002, pp. 847–852.

[58] J. Li and Q. Tan, "SmartInjector: Exploiting intelligent fault injection for SDC rate analysis," in *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT 2013)*. IEEE Computer Society Press, Oct. 2013, pp. 236–242.

[59] H. Schirmeier, C. Borchert, and O. Spinczyk, "Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors," in *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*. IEEE Computer Society Press, Jun. 2015.

[60] C. Dietrich, M. Hoffmann, and D. Lohmann, "Cross-kernel control-flow-graph analysis for event-driven real-time systems," in *Proceedings of the 2015 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '15)*. New York, NY, USA: ACM Press, Jun. 2015.

[61] J. Güthoff and V. Sieh, "Combining software-implemented and simulation-based fault injection into a single fault injection method," in *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing (FTCS '95)*. IEEE Computer Society Press, Jun. 1995, pp. 196–206.

[62] *DWARF Debugging Information Format Version 4*, DWARF Standards Committee, Jun. 2010. [Online]. Available: http://www.dwarfstd.org/doc/DWARF4.pdf

[63] D. Rath, "OpenOCD: Open on-chip debugging," Diploma Thesis, FH Augsburg, Jul. 2005.

[64] O. Spinczyk and D. Lohmann, "The design and implementation of AspectC++," *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, vol. 20, no. 7, pp. 636–651, 2007.