Noname manuscript No.
(will be inserted by the editor)

Simulation-based fault injection with QEMU for speeding-up dependability analysis of embedded software

Davide Ferraretto · Graziano Pravadelli

Received: date / Accepted: date

Abstract Simulation-based fault injection (SFI) represents a valuable solution for early analysis of software dependability and fault tolerance properties before the physical prototype of the target platform is available. Some SFI approaches base the fault injection strategy on cycle-accurate models implemented by means of Hardware Description Languages (HDLs). However, cycle-accurate simulation has revealed to be too time-consuming when the objective is to emulate the effect of soft errors on complex microprocessors. To overcome this issue, SFI solutions based on virtual prototypes of the target platform has started to be proposed. However, current approaches still present some drawbacks, like, for example, they work only for specific CPU architectures, or they require code instrumentation, or they have a different target (i.e., design errors instead of dependability analysis). To address these disadvantages, this paper presents an efficient fault injection approach based on QEMU, one of the most efficient and popular instruction-accurate emulator for several microprocessor architectures. As main goal, the proposed approach represents a non intrusive technique for simulating hardware faults affecting CPU behaviours. Permanent and transient/intermittent hardware fault models have been abstracted without losing quality for software dependability analysis. The approach minimizes the impact of the fault injection procedure in the emulator performance by preserving the original dynamic binary translation mechanism of QEMU. Experimental results for both x86 and ARM processors proving the efficiency and effectiveness of the proposed approach are presented.

Keywords Fault injection · Dependability analysis · Dynamic binary translation

D. Ferraretto, G. Pravadelli
Department of Computer Science, University of Verona, Italy
Tel.: +39 045 8027081
Fax: +30 045 8027068
E-mail: name.surname@univr.it

1 Introduction

Nowadays, software is more and more adopted to control safety-critical systems [28]. However, its escalating complexity added to the growing density of hardware platforms makes practically impossible to release perfect systems, even when sophisticated verification and validation approaches are adopted to guarantee their correctness [37]. In this context, the analysis of software dependability and fault tolerance properties are increasingly important aspects, highly recommended also by safety standards [34, 5, 21]. This especially applies to dependability analysis of embedded software (ESW) in the context of safety-critical applications, like for exampel, medical-surgical [30], e-commerce transactions [39], and community storage [41], besides well-known avionics, automotive, and nuclear power.

In order to evaluate the quality of dependability and fault tolerance approaches, fault injection and fault simulation techniques are widely adopted to observe how the system reacts in presence of faulty conditions [7]. In particular, these techniques are intended to test the behaviour of the application running on a real or virtual plant and to evaluate its tolerance to faults under a realistic set of input stimuli that mimic the final execution environment. In this context, four main categories of fault injection approaches exist: hardware fault injection (HFI), emulation-based fault injection (EFI), software fault injection (SWIFI) and simulation-based fault injection (SFI), with different advantages and drawbacks.

Hardware fault injection uses external physical sources to introduce faults into the system's hardware [14]. This represents the fastest approach, but also the most costly since it requires special-purpose HW. Moreover, it can be applied only when the hardware platform is available, and it presents a high risk of damaging the injected system. Emulation-based fault injection is based on the use of Field Programmable Gate Arrays (FPGAs) with the aim of providing performance similar to HFI approaches without their high cost and risk of damage [11]. However, EFI is classically limited to stuck-at fault injection and it requires a synthesizable model of the system under test.

Early analysis without the need of special-purpose HW can be performed by using SWIFI and SFI tools. SWIFI is based on the alteration of the software executing on the system under analysis [44]. SWIFI experiments can be carried on in near real time, but the software under test must be instrumented and faults cannot be injected into locations that are inaccessible to software. A non-intrusive approach with maximum amount of observability and controllability is represented by SFI, where a simulation model of the system under analysis, including the target processor, is implemented [26]. The main drawbacks in this case are represented by the large development effort required to model the simulator, and by poor simulation performance which drops proportionally by increasing the accuracy of the target processor model. Thus, one of the most challenging aspect of SFI approaches is to identify the most appropriate trade-off between accuracy and simulation performance.

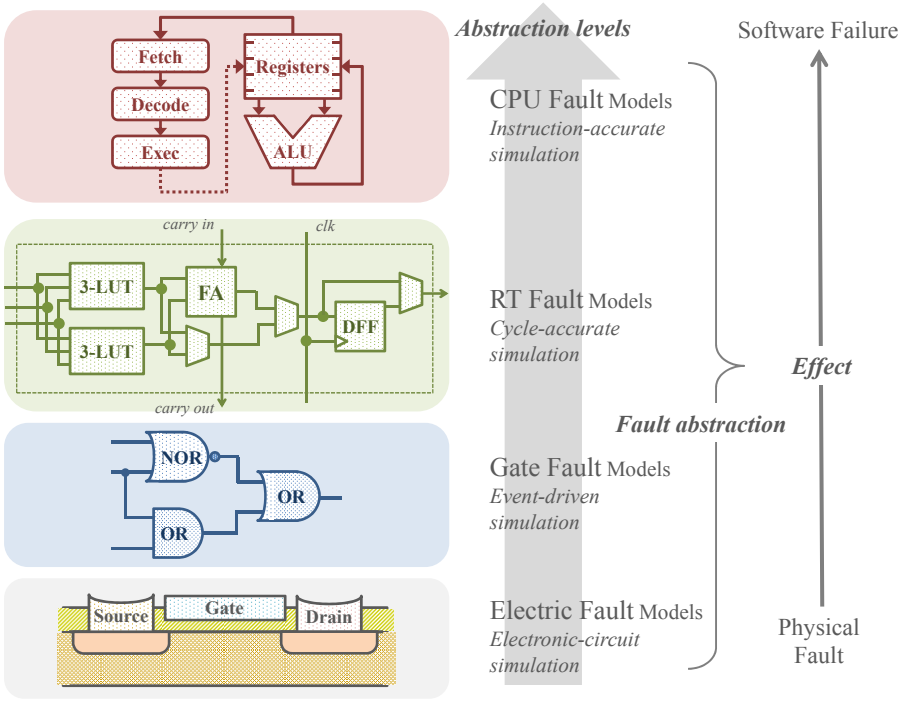

Fig. 1 CPU fault simulation taxonomy. Fault simulation at different abstraction levels is a trade-off between accuracy and simulation performance.

For example, Figure 1 depicts the taxonomy of CPU fault simulation at different abstraction levels. Fabrication errors, fabrication defects, and physical failures are collectively referred as physical faults [1], whose effect in the CPU hardware may manifest as a failure of the software executed on top of it. Fault models represent an abstraction of the physical faults on the behaviour of the simulated model. They can be defined at different abstraction levels, thus enabling different SFI approaches. The lowest levels provide more accurate results, but at the cost of a more time-consuming fault simulation, such that, for example in the case of electronic-circuit simulation, the purpose of evaluating a complete CPU and a software stack on top of it becomes impractical. Also traditional event-driven (gate-level) or cycle accurate (RTL) simulation is typically several orders of magnitude slower than real hardware. For example, in [42], and in other similar works, the authors defined a latch-accurate Verilog model of an ARM-compatible microarchitecture to study the effects of transient faults on memory cells. Such an environment, although highly accurate, can not be efficiently used for ESW dependability analysis due to the poor performance of hardware description language (HDL) simulators, which are usually far from being acceptable to simulate a complete CPU. Thus, higher levels of abstraction are preferred for meeting aggressive time-to-market targets. The fastest solutions are represented by purely func-

tional simulators that can almost reach the speed of the simulated hardware. However, simulating low-level faults could be very misleading when the simulation is only functional. According to these considerations, approaches based on instruction-accurate simulators, which rely on fast HW virtualization systems, have been recently proposed [29], with the aim of providing a fast, bust still quite accurate solution for SFI. Most of them (e.g., [4,31,12,2]) exploit the QEMU open source machine emulator, an efficient dynamic binary translation (DBT)-based virtualization system that allows to run a variety of unmodified guest operating systems on several CPU architectures [6]. In this context, main challenges, not completely addressed by the existing works (whose pros and cons are summarized in Section 2), are: emulating a wide category of faults by preserving their representativeness with respect to cycle-accurate CPU models [35], reducing the performance overhead due to fault injection to preserve as much as possible the efficiency of the DBT-based simulation, and being totally transparent form the user point of view.

Thus, the focus of the present work is to define an effective and efficient SFI approach based on QEMU that provides fast and realistic simulation of hardware faults for an early dependability analysis of ESW applications. Permanent, intermittent and transient faults in CPU registers of both CISC and RISC architectures are considered.

One key issue in our approach is the *fault representativeness*, i.e., the plausibility of the adopted fault model with respect to other fault models or physical faults. In this work, we pragmatically focus on the impact and consequences of injected faults on ESW, e.g., transient faults may affect running application by altering the data and execution flows. We have compared the behaviour of applications running both on our fault simulation environment and a HDL cycle-accurate simulator for representativeness assessing. In this case, the experiments using different fault simulation techniques have similar behaviours, thus the techniques can be considered equivalent from the point of view of ESW dependability analysis. The proposed approach exhibits the most suitable characteristics, i.e., simulation speed, and should be preferred.

In summary, the main contribution of the present work is the definition of a fully automatic fault simulation environment based on DBT for early ESW dependability analysis, which presents the following characteristics:

- the adaptation of traditional hardware fault models, i.e., stuck-at, transition, and transient fault models, to CPU instruction-accurate simulation; such fault models maintain the representativeness with respect to the corresponding fault models defined at register transfer level (RTL).
- the fault injection is not intrusive; it is necessary modify neither the CPU hardware where the application is executed on, nor the ESW that should be monitored;
- the fault simulation is deterministic, that is, each experiment run can be repeated for an arbitrary number of times and the executed application experiences exactly the same conditions;

– the fault simulation environment has been built on top of QEMU allowing cross-execution of ESW applications, i.e., engineers can develop on x86 host machine for a different target platform such as ARM and cross-validate the application thanks to the retargetable QEMU binary translation technology.

The rest of the paper is organized as follows. Related works are first summarized in Section 2. Then, Section 3 and Section 4 present, respectively, how the original QEMU execution flow has been modified to support fault injection, and how different kinds of faults are modelled inside QEMU. Section 5 deals with the set up of the simulation environment. Finally, Section 6 is devoted to experimental results, while Section 7 concludes the paper by drawing remarks.

2 Related works

Several works have been proposed in literature for injecting faults into *HDL models* of the systems. The adopted fault models range from low-level model of faults that may affect the physical circuits, e.g., stuck-at [16], transient [18], and delay faults [19], to faults modelling design errors, e.g., mutation testing [22]. Although an early analysis based on model simulation has its advantages, e.g., injecting faults into HDL descriptions is relatively straightforward, it provides variously reliable approximations of the systems and fault tolerance mechanisms and suffers of scalability problems due to HDL-simulation speed [17].

Fault injection on prototypes of the systems has proved to be a more realistic and fast solution to validate the final implementation. It ranges from HIF *physical* approaches that introduce faults into the hardware layer of the target system to SWIFI that perturbs the software layer for simulating physical faults.

Different approaches have been developed for injecting faults directly on the physical components of the systems. They affect the VLSI circuits by using heavy-ion radiations or electromagnetic interferences for perturbing the electronic status of the circuits [25, 24]; or they work at pin-level by directly bit flipping the pins of the circuit prototypes [3]. Although physical fault injection does not provide a perfect imitation of the original phenomena, it demonstrates to be a feasible solution. On the other side, it results to be expensive for both the required fault injection equipments and the development of fit-for-purpose prototypes.

SWIFI overcomes cost, controllability, and repeatability issues of physical fault injection. Such approaches usually modify the contents of registers and memory elements to emulate the effect of hardware faults [23, 9, 27, 8]. As a drawback, SWIFI tends to generate non significant experiments: the random injection on code and data segments in memory causes faults that remain latent throughout the experiments. Moreover, SWIFI techniques still require the target platform for accurately reproducing the faulty behaviours of the

actual hardware. Finally, SWIFI approaches are defined for either specific operating systems or application programming interfaces. For example, in [8] the authors propose a kernel-level fault injection approach. Such a kernel can be easily compiled for different hardware architectures, but the fault injection mechanisms have to be redefined for supporting different operating systems.

For addressing the previous speed, cost, reliability, and portability issues, SFI approaches have been proposed that rely on the use of virtualization systems based on Instruction Set Simulators (ISS) and, even preferably, SW emulators exploiting DBT. An ISS mimics the behaviour of a micro-architecture by executing instruction-by-instruction the target binary code. On the contrary, DBT works at basic block level guaranteeing a faster simulation [13]. A DBT-based emulator translates, at run time, all the instructions included in a basic block and caches the translated block for future uses. In addition to such a common feature, different optimizations can be implemented to further keep execution speed close to native execution, like, for example, dynamic recompilation where some parts of the code are recompiled to exploit information that are available only at run time (e.g., what portions of code are executed a large number of times).

One of the first work proposing the use of virtualization for simulation-based fault injection is UMLinux [38], a framework exploiting virtualization for testing networked machines running the Linux operating system in the presence of faults. A more generic approach, independent from the operating system, is proposed in [36]. This work relies on the FAUmachine, a fast open source virtual machine that supports fault injection capabilities, where transient, intermittent and permanent faults can be modelled into memory cells, disks and network. This approach is intended to simulate hardware as close to the corresponding physical hardware as possible thus affecting simulation speed. Moreover, fault injection in CPU registers is not permitted and it concentrates on a unique architecture, i.e., x86.

More recently, a few approaches relying on QEMU have been also proposed [10,4,31,12,2,15,20]. QEMU is a fast, open-source, machine emulator relying on DBT of the target CPU application code, which implements different optimizations to keep execution speed close to native execution. It started out as an emulator for x86 Linux binaries and, over the years, it has supported both emulation of most of the major CPU architectures (ARM, SPARC, etc.) and capabilities of system-emulation and virtualization. Moreover, extensions have been proposed for implementing timing and power analysis and co-simulation [33]. QEMU uses a 2-steps DBT: first the target machine code is translated into a generic intermediate representation. Then, this representation is in turn translated into the host machine code for execution. At the intermediate representation level, one can insert additional instrumentation operations, such as fault models and behavioural perturbations.

In [2] a QEMU-based soft error injection methodology is presented to accelerate the analysis and debugging of complex systems and facilitate the validation of fault tolerance techniques. However, the approach supports only x86 architectures and it considers only the bit flip fault models on general

purpose registers (GPRs). A more flexible approach based on QEMU is proposed in [10], where faults can be injected and triggered with a user defined probability inside CPU, RAM and peripherals. The target of this approach is to evaluate operating systems' susceptibility to faults. Another fault injection environment based on QEMU is presented in [4], but with a different purpose: creating a mutation based testing approach for binary code. The testing is seamlessly integrated at run-time into the binary translation cycle of QEMU. Thus, a way to inject high-level mutants (e.g., substitution of instructions, substitution of operands of an instruction, substitution of a condition on a branch, etc.) rather than faults on bits of registers is proposed, thus, this work targets SW design errors. Moreover, in [4] the instrumentation of a control flow graph extracted from the binary code is necessary, while our approach is not intrusive since it acts directly on the mechanism which emulates registers of the target CPU, independently from the executed code. QEMU is the basic environment also for the framework described in [20], which allows a system-level analysis of software countermeasures by featuring the simulation of high-level hardware faults targeting, for example, memory cells, register cells, or the correct execution of instructions. The framework is intended to support the generation of fault attack scenarios. In [31], the authors present a QEMU-based fault injection framework that requires a low overhead in comparison to a fault-free QEMU execution. However, fault representativeness is not mentioned in this work. An approach to abstract different kinds of hardware fault models inside QEMU that maintains the fault representativeness with respect to the corresponding fault models defined at RTL is, then, proposed in [12]. However, the accuracy achieved in this case is paid in terms of performance, since the approach requires to disable the caching mechanism implemented by QEMU. An enhancement of this work, which represents the basis for the methodology described in the next sections, is finally described in [15]. Our DBT-based approach has been implemented on top of QEMU. However, it applies to the more general concept of DBT independently from the specific emulator.

3 Methodology

An effective ESW dependability analysis should be based on an early evaluation of fault tolerance aspects of the applications. Thus dependability assessment should be performed in an efficient fault simulation environment. The proposed approach exploits instruction-accurate simulation and DBT to speed up simulation.

An instruction-accurate simulator provides an abstraction of the target CPU microarchitecture. Figure 2 compares the model diagram of a cycle-accurate ARM-compatible microarchitecture and the set of variables that store the CPU status during QEMU simulation. In this work, an ARM processor has been selected to exemplify the concepts, but our approach is independent from the specific processor.

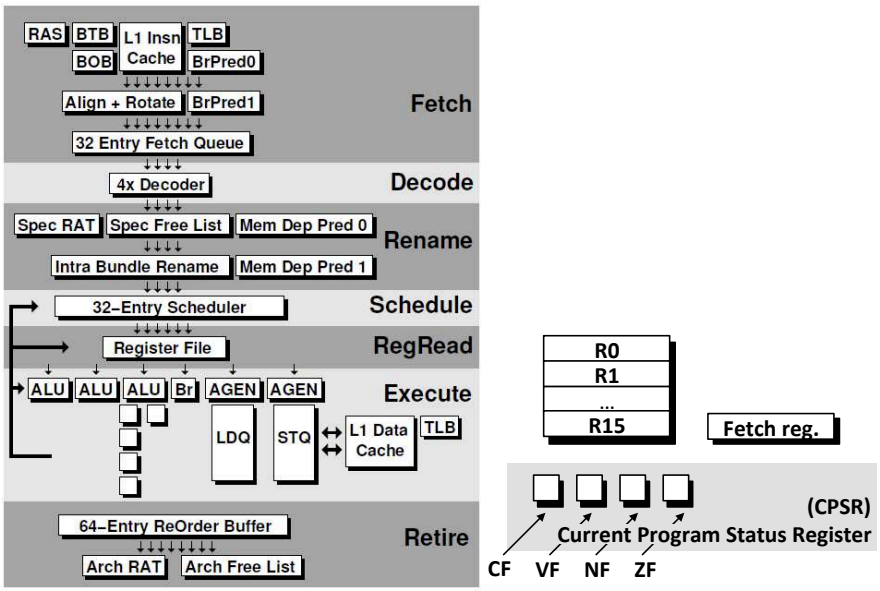

Fig. 2 (a) The model diagram of a cycle-accurate ARM-compatible microarchitecture [42]. (b) The eligible fault sites for a instruction-accurate model of an ARM microarchitecture (i.e., `CPUStatus` data-structure in QEMU source code).

Fault injection at gate level introduces faults into each gate, line, and memory element; while, fault injection at RTL introduces faults on more abstract components, e.g., multiplexers, ALU, and registers. In the case of an instruction-accurate CPU simulator, the faults are injected into the variables storing the CPU status (Figure 2.b). As a result, the faults at this abstraction level are a subset of all possible faults at gate level or RTL. But these faults directly affect the application execution during the dependability analysis and functionally subsume other faults that cannot be directly injected into the abstract model of the CPU. In particular, hardware faults whose effect propagates to CPU registers are covered by faults injected into the variables storing the CPU status, when an instruction-accurate CPU simulator is adopted as proposed in this paper. However, for the effect of propagation, a single hardware fault that cannot be directly modelled in the abstracted instruction-accurate model of the CPU, could correspond to a set of multiple faults inside the registers of the CPU.

QEMU works as depicted in Figure 3. When QEMU first encounters a piece of code, it converts target instructions to an intermediate code and then to a semantically-equivalent instruction sequence for the host machine, by calling the *gen_ intermediate_ code* function. In this function, the instruction corresponding to the program counter of the simulated CPU is fetched from the target binary code. The fetched instruction is then decoded into several micro-operations whose identifiers are concatenated into a micro-operations

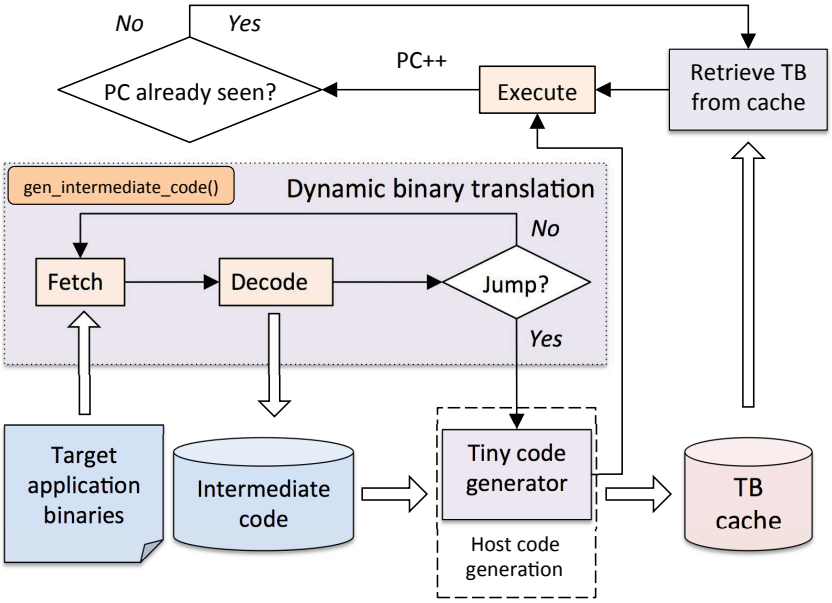

Fig. 3 The QEMU DBT mechanism.

identifier buffer. If the current instruction is not a jump, then the next target instruction is fetched and decoded. The binary translation stage ends after a jump instruction is decoded. The sequence of instructions collected between two jumps forms a Translation Block (TB). Once the block translation is finished, the *tiny code generator* of QEMU generates a host function composed of the concatenation of the micro-operation compiled code corresponding to the identifiers stored in the buffer. The resulted host function is executed and stored into an entry of the translation cache for future use. Then, the program counter is updated and QEMU verifies if the cache contains a TB corresponding to the new value of the program counter. If this is the case, such a block is directly executed. On the contrary, a new translation phase starts for converting the new piece of code.

In the context of modelling inside QEMU the presence of faults affecting the CPU registers, our main goals were:

1. avoiding instrumentation of the target binary application, such that the approach is totally transparent from the user point of view;
2. guaranteeing the effect of injected faults is the same we obtain by injecting them into a cycle-accurate implementation of the target CPU (e.g., an RTL model described by means of an HW description language like VHDL, Verilog or SystemC) to preserve fault representativeness;
3. reducing as much as possible the overhead introduced by the fault modelling procedure to preserve the high simulation speed offered by the DBT mechanism of QEMU.

Figure 4 depicts how the original DBT flow of QEMU has been modified to pursue the previous objectives. Possible locations for faults are GPRs, instruction register (IR) and program status register (PSR) that can be affected by permanent, transient and intermittent faults, as described in Section 4.

To achieve the first goal, fault injections is performed at fetching time for the IR, and at execution time for PSR and GPRs. In particular, for IR, we modified the DBT fetching mechanism of QEMU such that the presence of a fault in IR alters the way QEMU recognizes the instruction to be converted in the intermediate code. This causes that a different instruction with respect to the one stored in the IR is recognized and mapped on the instruction sequence for the host machine. On the contrary, for GPRs and PSR, we alter their values by manipulating the *CPUState* data structure QEMU uses to describe the status of the target CPU at execution time. In this way, fault injection is completely not intrusive from the target application point of view.

The second goal has been satisfied by imposing that QEMU operates in single step mode, i.e., each entry of the TB cache contains only one instruction. This reduces QEMU performance, but it is necessary to guarantee that the effect of the desired faults is active during the execution of each instruction of the target application, as it happens in traditional HDL-based cycle-accurate models. If a TB cache entry was composed of more than one instruction, only the first would be guaranteed to be affected by the presence of faults. In fact, the *CPUState* data structure, where we perform fault injection on GPRs and PSR, is atomically referred to each TB, such that no fault injection can be performed between two consecutive instructions belonging to the same TB. Thus, if the first instruction of a TB changes the value of the register where faults are injected, their effect is nullified for all the other instructions belonging to the same TB.

Finally, to achieve the third goal, the DBT-based approach of QEMU was almost completely preserved. A part from the adoption of the single step mode, only one change to the TB-caching mechanism was necessary to correctly simulate transient and intermittent faults. Since they are not permanent, a TB including instructions affected by transient faults cannot be stored in the cache. Otherwise, the same faults are simulated each time the same TB is executed, disrupting their transient nature (see Section 4.2). For all the other kinds of simulated faults, the caching mechanism is preserved.

4 Fault emulation

Our modified version of QEMU currently supports simulation of permanent, transient and intermittent faults. A permanent fault continues to exist until the faulty component is repaired. A transient fault occurs only once and we cannot trace it later on. If we repeat the operation, the fault goes away. An intermittent fault occurs not continuously but at irregular intervals. Several fault models have been proposed in the past for modelling such kinds of faults. Among the most popular, we selected the stuck-at [32] and the transition

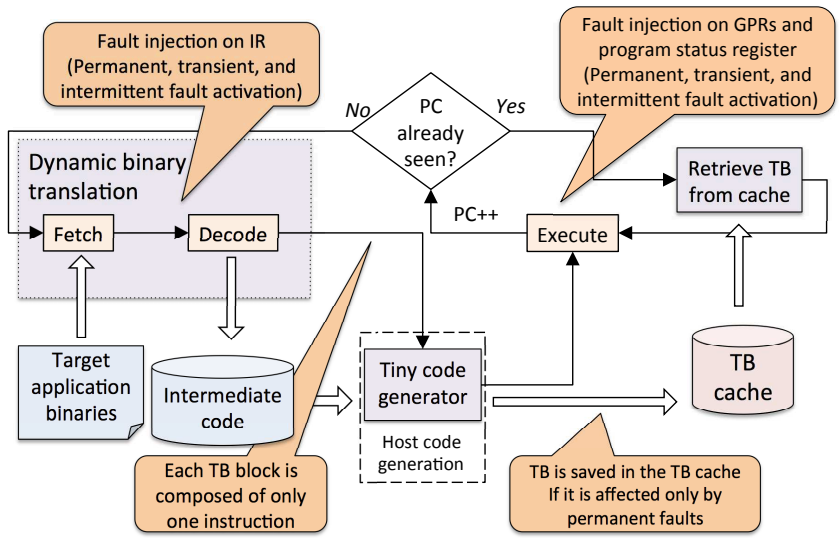

Fig. 4 Fault injection inside QEMU DBT mechanism.

fault [40] models as representative of permanent faults, and the bit flip fault model [43] for modelling transient and intermittent faults.

The stuck-at fault model assumes each bit of a register can be altered in two ways: stuck at the logic value 0 or stuck at the logic value 1. Each of these faults is called a single stuck-at 0 (SA0) or a single stuck-at 1 (SA1), respectively. Also in the transition fault model there are two possible faults associated to each bit of a register, i.e., slow-to-rise (STR) and slow-to-fall (STF). The effect of a STR (STF) consists of causing a delay in the switching activity of the affected bit, when it needs to pass from logic value 0 (1) to logic value 1 (0), such that the transition of the bit cannot be completed in time to guarantee the next instruction read its updated value. Finally, the bit flip fault model assumes each bit of a register can be switched to its opposite logic value or remain unaltered on the basis of a random probabilistic function. When bit flip is used for modelling transient fault, only a single activation is allowed. In case of intermittent fault simulation, the switching may happen repetitively at random intervals.

4.1 Permanent stuck-at fault simulation

Stuck-at faults are modelled by using a couple of bit vector masks per each CPU register, one for SA0 and one for SA1. Every mask reports the locations where faults must appear in its target register. Elements of a stuck-at 0 mask are set to 0 in correspondence of the bits to be altered with a SA0; the other elements are set to 1. Dually, elements of a stuck-at 1 mask are set to 1 in correspondence of the bits to be altered with a SA1; the other elements are set to 0. The length of the masks equals the size of the target registers. An

```
/* Use of a mask to inject stuck-at 0 faults in register number i of the target CPU. */
env->regs[i]&=mask_regerr[i].stuck_at_0;
/* Use of a mask to inject stuck-at 1 faults in register number i of the target CPU. */
env->regs[i]|=mask_regerr[i].stuck_at_1;
```

Fig. 5 Activation of stuck-at 0 and stuck-at 1 faults in GPRs. Variable *env* is a *CPUstate* data structure inside QEMU that stores the status of the CPU, including the value memorized in GPRs.

arbitrary number of faults can be concurrently activated in a mask, thus both single stuck-at and multiple stuck-at model can be adopted. The way masks are applied depends on three main aspects: the CPU architecture (i.e., CISC or RISC), the target register (i.e., GPRs, IR, or PSR), and how QEMU internally handles the value of registers.

4.1.1 ARM architecture

CPUs of the ARM family are based on a RISC architecture with a fixed-length instruction set architecture (ISA). Inside QEMU, GPRs of an ARM CPU are stored in the elements of the *regs* array included in the *CPUState* data structure. Thus, given a stuck-at fault mask, fault injection on general purpose registers is basically performed by executing at execution time the operations reported in Fig.5, i.e, bitwise *AND* and bitwise *OR* between the mask and the QEMU variable storing the value of the register, respectively, for stuck-at 0 and stuck-at 1.

On the contrary, inside QEMU there is not a specific variable devoted to emulate the instruction register. The instruction to be executed is retrieved by accessing directly the variable storing the value of the program counter in the *DisasContext* data structure, which includes information necessary to convert instructions into the QEMU intermediate code. No fault is injected in the program counter itself to avoid a probably immediate and irreparable block at the very beginning of the simulation. The presence of faults inside IR is then simulated by affecting the current instruction with masks at fetch time, as shown in Fig. 6.

Finally, after the execution of each instruction it is possible to simulate the presence of stuck-at faults in the program status register, called CPSR in ARM terminology. CPSR is read and written inside QEMU by means of *cpsr_read* and *cpsr_write* functions. After the execution of an instruction, CPSR value is retrieved by calling *cpsr_read*, affected by using a corresponding fault mask, and finally set by calling *cpsr_write*.

4.1.2 x86 architecture

CPUs of the x86 family are based on a CISC architecture with a variable-length ISA. Fault injection for GPRs is the same as for ARM architecture,

```
/* Use of a mask to inject stuck-at 0 faults in the fetched instruction. */

insn = arm_ldl_code(env, s->pc,
                    s->bswap_code);
insn &= mask_inserr_arm_stuck_at_0;
s->pc += 4;

/* Use of a mask to inject stuck-at 1 faults in the fetched instruction. */

insn = arm_ldl_code(env, s->pc,
                    s->bswap_code);
insn |= mask_inserr_arm_stuck_at_1;
s->pc += 4;
```

Fig. 6 Simulation of stuck-at 0 and stuck-at 1 faults in the ARM instruction register. The *arm_ldl_code* function fetches the instruction to be executed by accessing the program counter variable *pc* of the data structure *s*, which is of kind *DisasContext*.

while things are different to simulate faults inside the instruction register. The problem with CISC architectures derives from the fact that instructions can have different length. To address this issue, QEMU fetches an instruction one byte at time, by opportunely incrementing the program counter variable and by traversing a nested set of conditional statements, till the target instruction is finally recognized. At that point, the current instruction is converted into the QEMU intermediate code and the program counter points to the next instruction. Thus, there is not a variable inside QEMU storing the value of the IR where the stuck-at mask can be applied. This requires to solve two main issues: (i) injecting faults concurrently with the recognition of each byte of the target instruction; (ii) updating the value of the program counter variable to guarantee it correctly points to the next instruction independently from the effect of the faults injected in the current instruction.

The first issue is addressed by splitting a fault mask for the IR in a set of 8-bit sub-masks (for example, 4 sub-masks for a 32-bit i386 CPU). These are applied one by one to the result of the *ldub_code* function, which returns one by one the sequence of bytes of the target instruction, during the QEMU fetching steps. As a result, at the end of the fetching procedure, if the opcode of the target instruction is altered, QEMU will recognize and execute a different instruction with respect to the fault-free one.

The second issue required to recompute the program counter at the end of the fetching phase to correct a possible misalignment caused by the fault. A misalignment happens when the opcode of the original instruction is affected by a fault that causes QEMU recognizes an instruction of a different size. For example, opcode 46 (00101110 in binary), corresponding to the *INC* instruction, becomes opcode 62 (00111110 in binary), corresponding to the *BOUND* instruction, when a stuck-at 1 is activated in the fourth bit of the opcode. Since *BOUND* is a 2-byte long instruction, if QEMU recognizes *BOUND* instead of *INC*, due to the presence of the fault, the program counter variable is incremented by two bytes instead of one as required by *INC*, which is only one-byte long. Since we want that only the IR is affected by the fault, such a

```
/* Fetch of the instruction and fault injection is performed inside disas_insn. The
value of the program counter before fetching is stored in orig_pc. */

orig_pc = pc_ptr;
pc_ptr = disas_insn(dc, pc_ptr);

/* Now pc_ptr points to the next instruction according with the effect of faults. If
the executed instruction is not a jump alignment of the program counter could be
necessary. */

if (cnt_ins && !dc->is_jmp) {

/* target_disas is called to retrieve the size of the original instruction, which is then
stored in the global variable ret_bytes. */

  flag_disas = 1;
  target_disas(0, orig_pc, 1, 0);
  flag_disas = 0;

/* pc_ptr is realigned according to the value computed by target_disas. */

  pc_ptr = orig_pc + ret_bytes;
}
```

Fig. 7 Alignment of the x86 program counter after the simulation of faults in the IR.

misalignment of the program counter must be fixed. The solution we implemented extends the functionality of the *target_disas* function of QEMU such that it returns the size of the original instruction before it is changed by the injected fault(s). Then, after the execution of the faulty instruction the program counter can be realigned according to the value returned by *target_disas* as shown in Fig. 7. The re-alignment is not performed only in the case a fault causes the original instruction to become a jump. In this case, the program counter computed by QEMU is consistent with the location pointed by the jump without the need of a re-alignment.

Also the simulation of stuck-at faults in the x86 program status register, called EFLAGS in x86 terminology, is performed differently from ARM architecture. While in RISC architecture condition codes are updated only when explicitly requested, x86 sets them at the end of each instruction. Thus, for optimization purposes, QEMU adopts a lazy evaluation of x86 EFLAGS such that condition codes are computed only when needed. Instead of computing the condition codes after each x86 instruction, QEMU just stores one operand (CC_SRC), the result (CC_DST) and the type of operation (CC_OP). When the condition codes are needed, they can be calculated using this information. In such a context, fault simulation inside EFLAGS is computed by passing pointers to flag variables to a specific function (*flags_error*) which changes their values according to the selected fault mask.

4.2 Permanent transition fault simulation

Transition faults delay the switching of one or more bits in the affected registers. Inside QEMU, we simulate them by configuring the following set of parameters:

- *mask_delayerr*, a vector of unsigned integers whose elements memorize the transition fault mask for each CPU register. For every register i, *mask_delayerr*$[i]$ indicates which bits of i are affected by the delay (1 at position j indicates bit number j of register i is affected by a transition fault, while 0 means no fault is affecting such a bit);
- *delay_type*, a variable to annotate the kind of delay (i.e., STR or STF);
- *delay_length*, another variable to store the total duration of the delay affecting register bits;
- *delay_counter*, a matrix of integers to count the current duration of the active delay on every bit of every register. Element *delay_counter*$[i][j]$ reports the number of instructions to be executed before bit number j of register i is correctly switched, if a switching is caused by the currently-executed instruction.

Given the previous data structures, at the end of the execution of the current instruction, *mask_delayerr* is consulted and each bit j of each register i affected by a transition fault is updated as follows:

- if *delay_counter*$[i][j] = -1$ then the value of bit j of register i remains unchanged, even if it should be switched according to the effect of the current instruction, and *delay_counter*$[i][j]$ is reset to the *delay_length*. In this way, the update of bit j is delayed for *delay_length* instructions.
- if *delay_counter*$[i][j] > 0$, then *delay_counter*$[i][j]$ is decremented by 1 and the value of bit j of register i remains unchanged;
- if *delay_counter* $= 0$, then, finally, bit j of register i is set to 1 or 0 when *delay_type* is, respectively, STR or STF.

The strategy proposed for simulating transition faults applies in the same way for GPRs, IR and program status register, in both ARM and x86 architectures. However, in the case of IR and program status register the same arrangements described for stuck-at faults apply to avoid PC misalignment and to preserve lazy evaluation of condition codes.

4.3 Intermittent and transient bit flip fault simulation

To reflect the non permanent and non deterministic nature of intermittent and transient faults, bit flip activation depends on probabilistic values stored in the *p_reg_err* vector. Element *p_reg_err*$[i]$ memorizes the probability that, at run time, bit flips may affect register i in the CPU. Values in *p_reg_err* are decided by the user. Fault activation is then performed according to the following steps:

1. For each register i, a random number $n_i \in [0, 1]$ is generated, before the execution of each instruction;
2. n_i is compared with *p_reg_err[i]*. If n_i is lower than *p_reg_err[i]*, then bit flip faults are activated in register i by adopting the following procedure:
   (a) a second random number $m_i \in [0, 1]$ is extracted;
   (b) for each bit j of register i if $m_i \leq 0.5$ then j is set to 0, otherwise it is set to 1 independently from its original value.

According to the previous steps, in the current implementation, in average 50% of bits composing register i are altered when i is selected as location of transient faults. Thus, the probability an alteration happens on register i is $P(i) = p\_reg\_err[i] * 0.5 * register\_size$. By default, the effect of such alterations lasts for one instruction; however, a different duration can be set in the command line. If intermittent faults are simulated, the activation procedure is repeated for each instruction. On the contrary, a single instruction is affected by bit flip faults in a register, when the objective is to simulate a single occurrence of a transient fault.

The strategy proposed for simulating intermittent and transient faults applies in the same way for GPRs, IR and program status register, in both ARM and x86 architectures. However, in the case of IR and program status register the same arrangements described for stuck-at faults apply to avoid PC misalignment and to preserve lazy evaluation of condition codes.

It is also possible to configure the simulator such that a transient/intermittent fault becomes permanent once activated for the first time. In this case, TBs including affected instructions start to be saved in the cache as soon as the fault becomes permanent. To support this feature we extended the TB data structure such that it now includes also the register fault masks. This is necessary to avoid two different faults affecting the same instruction at different time instants are mapped, inconsistently, to the same entry in the cache when they become permanent. In this way, during QEMU execution, if a TB corresponding to the current program counter is present in the cache, but the related fault mask is different from the current one, the TB is discarded and regenerated.

5 Set up of the simulation

The fault injection approach presented in the previous section has been implemented into a fully automatic fault simulation environment based on DBT. It is intended for an early dependability analysis of embedded SW. The current implementation of the fault injector runs on the Linux operating system and it works for ARM and x86 architectures.

The set up of the simulation environment relies on a shell script that is in charge of making automatic the fault injection campaign and collecting the final results.

The user needs to define the target registers (GPRs, IR or PSR), the fault typology (stuck-at, transitions and bit flip), the corresponding fault masks (de-

fault masks can be used to inject all possible faults for all possible categories), and basic information required by the selected fault topology as reported in Sections 4.1, 4.2 and 4.3 (e.g., duration of the delay of transition faults, probability of activation for intermittent and transient faults) in a configuration file. Multiple faults can also be injected by configuring the fault masks with the desired number of fault locations.

Once configured, the tool runs the simulation by executing the target embedded SW upon QEMU. Results are collected in a set of log files, one per each execution run. Each log file reports the kind and location of the simulated fault(s), the result of the simulation, and the simulation time. The result of the simulation can be: *safe* when the benchmark responds as expected; *error* if an erroneous result is obtained; *loop* in case of a non-termination of the application; *crash* when a memory access violation causes to abort the execution. In case of crash, the log reports the last instruction that has been executed before the crash.

6 Experimental results

Experimental results have been conducted on three benchmarks (C code) generally used for testing the performance of CPUs: *btrees* that allocates and deallocates many binary trees thus stressing memory allocation, *mandelbrot* that exercises floating point operations to generate a Mandelbrot set, and *dhrystone* that is a synthetic program intended to be representative of integer programming.

Table 1 reports the number of injected faults per each category (stuck-at, transition and bit flip) on the different kinds of registers, i.e., general purpose register (*GPRs*), instruction register (*IR*) and program status register (*PSR*), for both x86 and ARM architectures. The numbers represent the possible way of altering registers by considering where stuck-at, transition and bit flip faults can be injected. Concerning stuck-at and transition faults, we injected one fault on a different bit of a different register per each run of the benchmarks. Thus, the number of injected faults, corresponds to the following expression:

$$\text{register size} * \text{kind of faults} * \text{number of registers}. \quad (1)$$

where, register size and number of registers is reported in Table 2[1], while the kind of faults are 2 for both stuck-at (SA0 and S01) and transition (STR and STF) faults. For bit flip, instead, it is not possible to know before execution the total number of injected faults, since it statistically depends on a probabilistic value (see Section 4.3). Thus, the numbers reported in Table 1 for the bit flip column correspond to the number of different probabilistic values we use for deciding when activating bit flipping on the whole register multiplied for the number of registers of each category. This means that, for bit flip faults, a

[1] For the program status register, we considered 4 flags in case of ARM (CF, VF, NF, ZF) and 6 flags in case of x86 (CF, PF, AF, ZF, SF, OF).

CPU	# stuck-at			# transition			# bit flip		
	GPRs	IR	PSR	GPRs	IR	PSR	GPRs	IR	PSR
x86	512	64	12	512	64	12	80	10	10
ARM	1024	64	8	1024	64	8	160	10	10

Table 1 Number of faults injected in CPU registers.

CPU	GPRs		IR		# PSR	
	size	number	size	number	size	number
x86	32	8	32	1	6	1
ARM	32	16	32	1	4	1

Table 2 Characteristics of CPU registers.

Benchmark	CPU	# Stuck-at	# Transition	# Bit flip	# Total
btrees	x86	4,517,470	2,722,814	7,298	7,247,582
	ARM	26,741,522	18,065,640	2,827,437	47,634,599
mandelbrot	x86	4,852,735	56,213	588,104	5,497,052
	ARM	59,587,175	19,555,500	2,876,513	82,019,188
dhrystone	x86	14,340,698	3,780,511	9,830	18,131,039
	ARM	14,397,994	23,138,774	2,486,585	40,023,353

Table 3 Number of activated faults during execution.

Benchmark	x86		ARM	
	Fault-free	Faulty	Fault-free	Faulty
btrees	71.1	169.5	20.0	339.4
mandelbrot	61.6	189.0	108.6	923.36
dhrystone	318.1	825.9	66.86	986.15

Table 4 Simulation time (seconds).

random number of bits of a single register has been altered per each run of the benchmarks. The actual number of fault activations occurred during the execution of the benchmarks, according to the injected faults, is reported in Table 3. These values depend on the number of times each register has been used by the benchmark instructions during benchmark execution.

Simulation times are shown in Table 4, where columns *faulty* report the time required to fault simulate the benchmarks, while columns *fault-free* show the time required to simulate them without faults (i.e., by using the original version of QEMU). For a fair comparison, the fault-free time has been obtained by executing the original version of QEMU as many times as required to simulate all the faults, as reported in Table 1. We can observe that the overhead induced by the fault injection mechanism with respect to the fault-free simulation is substantially acceptable considered the huge number of fault activations reported in Table 3. Fault simulation for x86 resulted to be faster, even if for fault-free simulation the opposite is true. This depends on the fact that the emulated x86 architecture has only 8 GPRs while ARM architecture

Benchmark	*RTL*	*ISS*	*DBT w/o cache*	*DBT with cache*
btrees	20.51	2.16	0.72	0.14
mandelbrot	432.64	51.79	0.69	0.39
dhrystone	108.42	16.54	0.46	0.42

Table 5 Comparison of fault simulation time (seconds) for ARM architecture by considering RTL cycle-accurate fault simulation, ISS-based fault simulation, DBT-based simulation with disabled cache and DBT-based simulation with the cache enabled.

has 16 of them, thus the total number of fault activations is much higher for ARM than x86 (see Column *Total* of Table 3).

A further comparison concerning simulation time is reported in Table 5 for the specific case of the ARM CPU. It shows the average time required for simulating the benchmarks in presence of one of the possible injected fault by comparing four different approaches:

1. cycle-accurate fault simulation performed on an RTL Verilog model of an Amber ARM[2] (corresponding to the version emulated in QEMU) by using Modelsim (column *RTL*);
2. fault simulation on QEMU acting as a traditional ISS without applying DBT (column *ISS*);
3. fault simulation on QEMU exploiting DBT with the cache disabled as proposed in [12] (column *DBT w/o cache*);
4. fault simulation on QEMU exploiting DBT and without disabling the cache mechanism as proposed in this paper (column *DBT with cache*).

As a result, preserving the use of the cache in the DBT mechanism, as proposed in this paper, provides a significant speed-up with respect to the approach proposed in [12], respectively 5.04x, 1.77x and 1.11x for *btrees*, *mandelbrot* and *dhrystone*.

A comparison on simulation time with the other state-of-the art tools referred in Section 2 would be also interesting. Unfortunately, this was not possible, either because the tool is not freely available, or because it targets different fault types/locations with respect to our approach. However, we think all approaches based on DBT have comparable execution times. We are not expecting differences of orders of magnitudes among different DBT-based approaches. The real improvement is with respect to more detailed models and accurate simulation (e.g., at RTL), as confirmed in Table 5, where we compared the speed of our approach (last column) with the speed of RTL fault simulation (second column).

Table 6 and Table 7 report the effect of injected faults on the execution of benchmarks, respectively, for x86 and ARM. As expected, crashes are the most common result for all kinds of faults, and, in particular, for bit flips since they alter several bits of a register simultaneously.

[2] RTL model of x86 was not available in our laboratory, thus this kind of comparison has not been conducted for x86.

Effect	*btrees*			*mandelbrot*			*dhrystone*		
	stuck-at	*transition*	*bit flip*	*stuck-at*	*transition*	*bit flip*	*stuck-at*	*transition*	*bit flip*
loop	2%	1%	0%	4%	1%	0%	5%	3%	0%
safe	10%	34%	0%	15%	48%	0%	11%	34%	0%
error	7%	8%	0%	9%	11%	0%	12%	9%	0%
crash	81%	58%	100%	72%	40%	100%	72%	54%	100%

Table 6 Effect of faults on x86.

Effect	*btrees*			*mandelbrot*			*dhrystone*		
	stuck-at	*transition*	*bit flip*	*stuck-at*	*transition*	*bit flip*	*stuck-at*	*transition*	*bit flip*
loop	5%	2%	1%	8%	7%	1%	5%	3%	4%
safe	16%	36%	0%	23%	53%	0%	19%	39%	0%
error	12%	16%	0%	8%	9%	2%	14%	16%	0%
crash	67%	46%	99%	60%	32%	98%	62%	42%	96%

Table 7 Effect of faults on ARM.

As a final experiment, Figure 8 reports the fault simulation results for the proposed approach (*QEMU*) with respect to the cycle-accurate RTL simulation of the Amber ARM (*RTL*). For each benchmark and for each fault site the responses of the ESW to fault injections are reported. The fault sites are grouped in general purpose registers (*GPRs*), program status register (*PSR*), and instruction register (*IR*). Numbers in the bars refer to the percentage of faults that cause safe, error, loop and crash responses. In general, we can notice that the trends between our approach and the RTL cycle-accurate one are similar. The difference between them is at most 3%.

Note that the proposed approach does not correlate the software failures with the physical faults, as well it does not provide the designer with timing and topological information that may characterize the faults at a lower abstraction level. But, on the basis of the experimental results, we can affirm that dependability analysis of ESW can be carried out at an earlier phase of the design cycle, when the application and the proposed instruction-accurate fault simulator are available. In fact, the dependability analysis gains significant speed up, but maintains the accuracy of the cycle-accurate simulation.

7 Conclusions

In this paper we presented an automatic non-intrusive simulation-based fault injection framework based on QEMU. Three different kinds of fault can be simulated according with widely adopted fault models. The efficiency of the approach is confirmed by experimental results for both x86 and ARM architectures. Future works will be devoted to extend the approach with further fault models and to compare its performance and accuracy with respect to fault injection approaches working on FPGA.

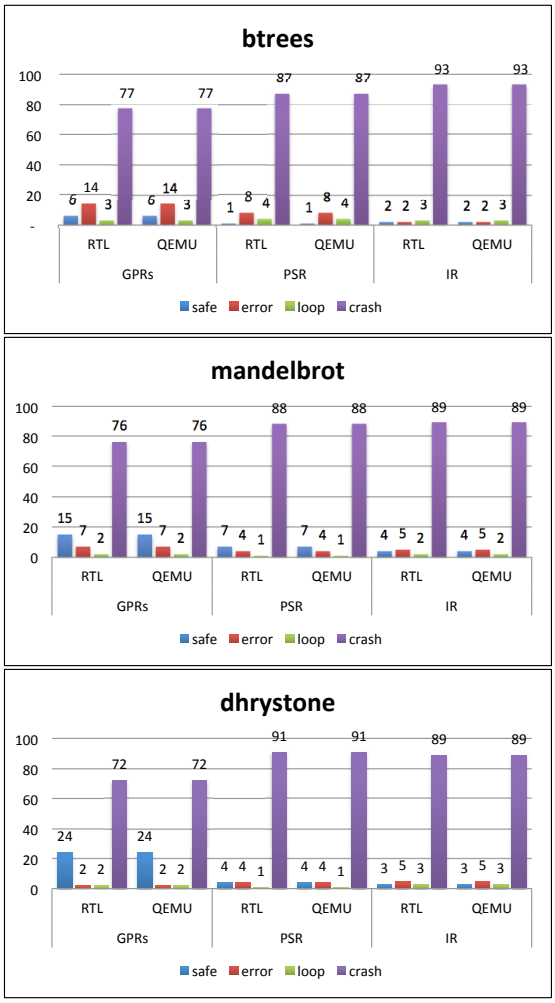

Fig. 8 Comparison of fault simulation results for the ARM architecture.

References

1. Abramovici, M., Breuer, M., Friedman, A., of Electrical, I., Engineers, E.: Digital systems testing and testable design, vol. 2. Computer Science Press, New York (1990)
2. de Aguiar Geissler, F., Lima Kastensmidt, F., Pereira Souza, J.: Soft error injection methodology based on qemu software platform. In: Proc. of IEEE LATW, pp. 1–5 (2014)
3. Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J., Laprie, J., Martins, E., Powell, D.: Fault injection for dependability validation: A methodology and some applications. IEEE Transactions on Software Engineering **16**(2), 166–182 (1990)
4. Becker, M., Baldin, D., Kuznik, C., Joy, M.M., Xie, T., Mueller, W.: Xemu: An efficient qemu based binary mutation testing framework for embedded software. In: Proc. of ACM EMSOFT, pp. 33–42 (2012)
5. Bell, R.: Introduction to iec 61508. In: Proc. of SCS, SCS '05, pp. 3–12 (2006)

6. Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proc. of USENIX ATEC (2005)
7. Benso, A., Prinetto, P. (eds.): Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation. Springer (2003)
8. Cabodi, G., Murciano, M., Violante, M.: Boosting software fault injection for dependability analysis of real-time embedded applications. ACM Transactions on Embedded Computing Systems **10**(2), 24 (2010)
9. Carreira, J., Madeira, H., Silva, J.: Xception: A technique for the experimental evaluation of dependability in modern computers. IEEE Transactions on Software Engineering **24**(2), 125–136 (1998)
10. Chylek, S., Goliszewski, M.: Qemu-based fault injection framework. Studia Informatica **33**(4), 25–42 (2012)
11. Civera, P., Macchiarulo, L., Rebaudengo, M., Sonza Reorda, M., Violante, M.: Exploiting fpga for accelerating fault injection experiments. In: Proc. of IEEE IOLTS, pp. 9–13 (2001)
12. Di Guglielmo, G., Ferraretto, D., Fummi, F., Pravadelli, G.: Efficient fault simulation through dynamic binary translation for dependability analysis of embedded software. In: Proc. of IEEE ETS, pp. 1–6 (2013)
13. Ebcioglu, K., Altman, E., Gschwind, M., Member, S., Member, S., Sathaye, S.: Dynamic binary translation and optimization. IEEE Transactions on Computers **50**, 529–548 (2001)
14. Entrena, L., López-Ongil, C., García-Valderas, M., Portela-García, M., Nicolaidis, M.: Hardware fault injection. In: M. Nicolaidis (ed.) Soft Errors in Modern Electronic Systems, *Frontiers in Electronic Testing*, vol. 41, pp. 141–166. Springer (2011)
15. Ferraretto, D., Pravadelli, G.: Efficient fault injection in QEMU. In: Proc. of IEEE Latin American Test Symposium (LATS) (2015)
16. Fin, A., Fummi, F., Pravadelli, G.: Amleto: A multi-language environment for functional test generation. In: Proc. of IEEE International Test Conference (ITC), pp. 821–829 (2001)
17. Gil, D., Baraza, J., Gracia, J., Gil, P.: Vhdl simulation-based fault injection techniques. In: Fault injection techniques and tools for embedded systems reliability evaluation, pp. 159–176. Springer (2004)
18. Gil, D., Gracia, J., Baraza, J.C., Gil, P.J.: A study of the effects of transient fault injection into the vhdl model of a fault-tolerant microcomputer system. In: Proc. of IEEE International On-Line Testing Workshop (IOLTW), pp. 73–79 (2000)
19. Guarnieri, V., Fummi, F., Chakrabarty, K.: Reduced-complexity transition-fault test generation for non-scan circuits through high-level mutant injection. In: IEEE Asian Test Symposium (ATS) (2012)
20. Holler, A., Krieg, A., Rauter, T., Iber, J., Kreiner, C.: QEMU-based fault injection for a system-level analysis of software countermeasures against fault attacks. In: Proc. of Euromicro Conference on Digital Systems Design (DSD) (2015)
21. International Organization for Standardization: Product development: Software level. ISO 26262-6 (2011)
22. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering **37**(5), 649–678 (2011)
23. Kanawati, G., Kanawati, N., Abraham, J.: Ferrari: A flexible software-based fault and error injection system. IEEE Transactions on Computers **44**(2), 248–260 (1995)
24. Karlsson, J., Folkesson, P., Arlat, J., Crouzet, Y., Leber, G., Reisinger, J.: Application of three physical fault injection techniques to the experimental assessment of the mars architecture. In: Proc. of IFIP Working Conference on Dependable Computing for Critical Applications, pp. 267–287 (1995)
25. Karlsson, J., Liden, P., Dahlgren, P., Johansson, R., Gunneflo, U.: Using heavy-ion radiation to validate fault-handling mechanisms. IEEE Micro **14**(1), 8–23 (1994)
26. Kooli, M., Di Natale, G.: A survey on simulation-based fault injection tools for complex systems. In: Proc. of IEEE DTIS, pp. 1–6 (2014)
27. Krishnamurthy, N., Jhaveri, V., Abraham, J.: A design methodology for software fault injection in embedded systems. In: Proc. of IFIP International Workshop on Dependable Computing and its Applications, pp. 12–14 (1998)

28. Larrucea, X., Combelles, A., Favaro, J.: Safety-critical software [guest editors' introduction]. IEEE Software **30**(3), 25–27 (2013)
29. Le, M., Tamir, Y.: Fault injection in virtualized systems - challenges and applications. IEEE Trans. on Dependable and Secure Computing **PrePrints** (2014). DOI 10.1109/TDSC.2014.2334300
30. Leveson, N., Turner, C.: An investigation of the Therac-25 accidents. Computer **26**(7), 18–41 (1993)
31. Li, Y., Xu, P., Wan, H.: A fault injection system based on QEMU simulator and designed for BIT software testing. In: Proc. of ISCCCA (2013)
32. McCluskey, E., Tseng, C.W.: Stuck-fault tests vs. actual defects. In: Proc. of IEEE ITC, pp. 336–342 (2000)
33. Mueller, W., Pétrot, F.: 1st International QEMU Users' Forum. Grenoble (2011)
34. NASA: NASA software safety guidebook. NASA-GB-8719.13 (2004)
35. Natella, R., Cotroneo, D., Duraes, J., Madeira, H.: On fault representativeness of software fault injection. IEEE Trans. on SW Eng. **39**(1), 80–96 (2013)
36. Potyra, S., Sieh, V., Cin, M.D.: Evaluating fault-tolerant system designs using faumachine. In: Proc. of ACM EFTS (2007)
37. Seong, P.H. (ed.): Reliability and Risk Issues in Large Scale Safety-critical Digital Control Systems. Springer (2009)
38. Sieh, V., Buchacker, K.: Umlinux - a versatile swifi tool. In: Proc. of EDCC, pp. 159–171 (2002)
39. Team, A.S.: Amazon S3 availability event: July 20, 2008 (2008). URL http://status.aws.amazon.com/s3-20080720.html
40. Waicukauski, J., Lindbloom, E., Rosen, B.K., Iyengar, V.: Transition fault simulation. IEEE Design Test of Computers **4**(2), 32–38 (1987)
41. Wang, F., Agrawal, V.: Soft error considerations for computer web servers. In: In the Proc. of Southeastern Symposium on System Theory, pp. 269–274. IEEE (2010)
42. Wang, N., Quek, J., Rafacz, T., Patel, S.: Characterizing the effects of transient faults on a high-performance processor pipeline. In: Proc. of IEEE International Conference on Dependable Systems and Networks, pp. 61–70 (2004)
43. Yount, C., Siewiorek, D.: A methodology for the rapid injection of transient hardware errors. IEEE Trans. on Computers **45**(8), 881–891 (1996)
44. Yuste, P., Ruiz, J., Lemus, L., Gil, P.: Non-intrusive software-implemented fault injection in embedded systems. In: R. de Lemos, T. Weber, J. Camargo JoãoBatista (eds.) Dependable Computing, *Lecture Notes in Computer Science*, vol. 2847, pp. 23–38. Springer (2003)