Abstract—Embedded systems with high dependability requirements have to satisfy ever-growing demands for high computing performance, number of implemented features and cost-efficiency. This leads to a move to commercial off-the-shelf hardware components. At the same time, the number of faults in embedded systems increases due to shrinking hardware feature sizes and increasing software complexity.

In order to create dependable software-intensive systems, we propose reliability-aware software development to create systems with high inherent fault masking capabilities. Therefore, appropriate fault injection techniques for assessing the fault tolerance at various development stages are needed. Furthermore, we investigate in diverse compiling as a cost-efficient method to efficiently exploit redundancy. Finally, we propose to examine potential benefits of dynamic automated diversity techniques proposed to increase the security of systems, regarding their capabilities to increase the fault tolerance regarding common-cause faults in redundant systems.

I. INTRODUCTION

Since ever more functionalities are integrated into electronic devices, embedded systems have to fulfill increasing demands on high computing performance and have to implement an ever wider range of features. This leads to a trend to commercial off-the-shelf (COTS) hardware. While there are processors available that are designed and produced for high reliability applications, their performance typically lags behind that of COTS multi-purpose processors. Additionally, COTS processors typically come at a much lower price than their reliability-hardened counterparts [1]. However, using COTS processors in dependable systems poses many challenges. Frequently these processors do not provide basic built-in self-diagnose features such as hardware-based error correction. Furthermore, hardware implementation details, such as netlists, or register-transfer-level models are typically not available, if software and hardware providers are separate entities.

At the same time, the probability that faults jeopardize the dependability of embedded systems is increasing. Due to shrinking feature sizes, hardware is getting ever more vulnerable to operational faults, such as soft errors due to radiation and permanent errors due to manufacturing, process variations, aging, etc. Furthermore, the number of software-faults is increasing due to growing software complexity. Additionally, ever more systems are interconnected, which enhances the opportunities for malicious attacks. To sum it up, faults cannot be completely prevented, so they remain in every complex system. Consequently, fault tolerance is required in order to handle faults during operation. In this work, we propose fault tolerance methods in order to increase dependability, which is a superordinate concept regrouping different attributes such as reliability, availability, safety, maintainability etc. We focus on reliability, safety and security. As defined in [2] reliability denotes the continuity of correct service and safety the absence of catastrophic consequences. Security implies confidentiality, integrity and availability.

We propose a software-based approach towards hardware fault tolerance in order to enhance the dependability. However, such software-implemented methods are not well suited to handle transient hardware faults affecting the program execution (e.g. cause control flow errors), which occur frequently in COTS-based systems [3]. The consequences of such faults are in most cases that the faulty unit stops or enters an infinite loop and no bad output is produced. Such fail-stop faults can be detected relatively easily by typical fault tolerance techniques, such as a watchdog. However, there are also Byzantine faults, where the system continues to run but produces incorrect outputs. We focus on software-based methods to identify these Byzantine faults, since they are much harder to detect.

Fig. 1 shows the suggested approach and contributions in order to increase the reliability of systems without introducing much implementation and development overhead. First, we propose to efficiently exploit inherent hardware fault masking capabilities at software and application level. Furthermore, we investigate methods to efficiently exploit redundancy. Since semiconductors are increasingly integrated, there are ever more opportunities to establish redundancy. For example, the trend to multi-core processors in the embedded domain offers new possibilities to use inherent spatial redundancy. Additionally, idle times can be exploited to realize time redundancy. To achieve these objectives, appropriate development methods are required. In this work, we suggest reliability-aware development using advanced fault injection (FI) techniques and automated diversity as such methods.
These methods can inject a variety of faults in hardware that modifies the state of the system (e.g. [7], [8], [9]). However, these techniques often require a modification of the source code and it is hard to model permanent faults.

**B. Virtual Fault Injection**

To overcome the limitations of traditional FI techniques regarding COTS-based systems described above, there are proposals to adapt emulators performing hardware virtualization to simulate faults at system-level. The Quick EMUlator (QEMU) [12] is open source and targets the emulation of hardware for embedded systems. It features the fast emulation of several CPU architectures (e.g., ARM, x86, Sparc, Alpha) on several host platforms (e.g., ARM, x86, PowerPC). Recently, tools realizing the injection of soft errors using QEMU have been proposed (e.g. [13], [14]). In addition to soft errors, permanent memory-related faults are considered in [15].

However, we propose a QEMU-based VFI framework that supports an enhanced fault model that goes beyond memory cell faults (see Table I). In [10], we have shown that this allows to fulfill IEC 61508 SIL 3 requirements regarding fault modeling to assess fault tolerance techniques for processor and RAM [10]. According to their duration, the simulated faults can be permanent, transient, and intermittent. The fault can be triggered if a certain program counter is reached, after a specific time, or whenever the victim component is accessed. Furthermore, the accuracy of the fault models is increased by taking particularities of memory components into account. The framework features functional fault models describing the deviation of an observed and a specified behavior after a certain number of memory operations have been performed.

**III. Automated Diversity**

A simple addition of redundancy is still vulnerable to faults affecting all redundant channels. Diversity is needed to detect common-cause faults such as software bugs that exist in every redundant instance or hardware faults affecting all redundant channels. For example, if two cores of a multicore system are used as redundant channels, both calculations could be affected from a fault in a shared resource such as the RAM. Furthermore, special attention to common hardware faults is required when time redundancy is applied in only one hardware channel. The goal of diversity methods is to increase the probability that the consequences of a fault in the diverse variants are different, so that it can be detected [16].

A classic approach to add diversity is N-version programming meaning that several development teams work independently to design and implement N software versions. However, since this approach is very cost-intensive, techniques to automatically introduce diversity have been proposed.

Recently, automated diversity gained attention in the security domain as a technique of diversifying each deployed program version [17]. This forces attackers to target each system individually. Preliminary results described in [18] also indicate that automated diversity increases the resilience regarding hardware faults in redundant systems. Whereas static techniques generate several diverse execution variants before deployment, static techniques generate one program that can be diversified in operation [19].

---

**Fig. 2: Reliability-aware development process using FI tools in multiple stages to achieve early feedback about fault tolerance characteristics. Adapted from [4].**

**II. Fault Injection for Reliability-Aware Software Development**

During software development, the impact of underlying hardware faults is typically neglected. However, the vulnerability to hardware faults can significantly be reduced by exploiting inherent fault masking properties of software algorithms and by a defensive programming approach (e.g. plausibility checks). In order to cost-efficiently increase the hardware fault tolerance, we propose to integrate reliability-awareness in various software development stages. The aim is to close fault tolerance issues as early as possible before physical hardware testing is performed. This poses the need for FI frameworks that are applicable in various development stages.

In [4], we propose to integrate FI during model checking in early design stages (see Fig. 2). This supports to design algorithms in such a way that high-level fault masking properties are enhanced. Additionally, we present a Virtual FI (VFI) framework to assess the fault tolerance of software without the need for detailed hardware models. The framework can be applied by software programmers and test engineers to assess the impact of hardware faults on the behavior of software at different levels (e.g. individual functions/modules, completely integrated software system, etc.).

**A. Fault Injection Background**

Most established FI techniques, such as introducing faults in the finalized parts or simulation- and emulation-based techniques require a detailed design of the processor (e.g. hardware layout, RTL model, or netlist) [5]. However, this information is typically not available when using third-party processors.

An approach to perform FI for COTS processor-based platforms is to use their on-chip debug features as proposed in [6]. However, these approaches are very platform-dependent, since they heavily rely on specific hardware components such as JTAG-devices. Another approach is software-based FI based on the execution of additional software on the target platform that modifies the state of the system (e.g. [7], [8], [9]). These methods can inject a variety of faults in hardware components, which are accessible by the software, such as registers, memory, etc. However, these techniques often require a modification of the source code and it is hard to model permanent faults.
A. Static Automated Software Diversity

A cost-efficient approach is to exploit differences of off-the-shelf compilers to create diverse binaries. Intuitively, the diverse usage of compilers enhances the chance of detecting and tolerating programmatic faults of the compiler. Furthermore, compilers provide a wide range of optimizations, such as speculative branch prediction and the production of branch-free code. These optimizations enable diverse behaviors, even if the same source code base is used.

1) Diverse Compiling for Hardware-Fault Tolerance: Time redundancy means to execute the same calculation multiple times subsequently on the same hardware. Then, diversity is needed in order to detect permanent hardware faults. We evaluated the efficiency of diverse compiling for detecting such permanent hardware faults. We experimentally quantified the efficiency of diverse compiling by using applications from the MiBench benchmark suite [20]. The C source code was compiled with the GCC and Clang compiler using different optimization flags. Then, we used the proposed VFI framework described above to introduce permanent hardware faults. In total, we injected 165,888 register faults and 100 inactive decoder faults during the execution of the applications. Our results show that the combination of different optimization levels allows to detect up to 92% of all register cell faults and 73% of all instruction decoder faults. This yields the conclusion that diverse compiling is a low-cost solution that is well suited to significantly reduce the vulnerability to permanent processor faults.

2) Diverse Compiling for Software-Fault Tolerance: In [21], we show that diverse compiling not only increases the chance to detect hardware faults, but also enhances the software fault tolerance. The memory is organized differently, when using different compilers and compiler flags. This enhances the chance of detecting memory-related software bugs, such as missing memory initialization, during runtime. We used the same applications and compiler combinations as described above. Then we injected typical programming mistakes by mutating the source code with a software-fault injection tool described in [22]. In total, 296 representative memory-related faults were injected. Our results show that the efficiency of diverse compiling depends on the used compiler combinations, the application and the processed input value. However, the experiments also clearly indicate that diverse compiling improves the chance of finding software programming bugs by detecting up to about 70% of all injected memory-related software bugs.

However, the experimental investigation has also shown that the success of diverse compiling depends on the application, the workload of the application and the processor architecture. For example, when combining the GCC compiler with the highest optimization level (O3) and the Clang compiler with the lowest optimization level (O0), 83% of all software faults introduced in a GSM benchmark have been detected. However, this coverage is only as high as 36% when considering an image processing application (Susan MiBench). Similar variations have been observed when considering the detection of hardware faults. Thus, in the future, we plan to further evaluate the influence of software characteristics on the efficiency of diverse compiling in order to offer a methodology to chose those replicas that offer the best resilience.

B. Dynamic Automated Software Diversity

The idea of dynamic diversity is to integrate randomization points in the software that can be changed during execution. Then, one program can perform diverse executions leading to the same results. Diversity in execution can mean, for example, diverse performances, diverse memory locations, or diverse orders of execution. The only established dynamic diversity approach to increase the reliability is data re-expression to obtain data diversity [23]. Re-expression algorithms transform the input to produce new inputs to redundant variants. After program execution the distortion introduced by the re-expression is removed in order to obtain the intended output. The goal of this technique is that a given initial data within the program failure region should be re-expressed to an input data that exists outside that failure region.

Recently, multiple automated diversity techniques have also been proposed to improve the security of software-intense systems [19] as exemplified in Table II. However, potential benefits of these approaches to also tolerate non-malicious faults have not been examined so far. Thus, we propose to increase the cross-fertilization between the security and reliability research regarding dynamic software diversity. We plan to use selected techniques to introduce diversity in redundant program variants in order to increase the fault tolerance regarding common-cause faults. For example, binary stirring rewrites binaries in such a way that a new program encoding is used each time a program is loaded. Although this approach has been proposed to prevent code injection attacks, we assume that it could also be adapted in order to increase the reliability. For instance, processor faults affecting certain instructions could be circumvented by performing the calculations with different instructions.

The key advantage of dynamic diversity compared to static techniques, is the ability to adapt the execution during runtime in order to achieve resilience. Faults could be proactively mitigated by changing the setting of the dynamic randomization. For example, the memory layout could be changed in such a way that a detected address decoder fault is bypassed.

<table>
<thead>
<tr>
<th>Comp.</th>
<th>Target</th>
<th>Fault modes</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Instruction decoder</td>
<td>New value</td>
<td>Replaces current instruction with a given instruction</td>
</tr>
<tr>
<td>Register</td>
<td>Register cell</td>
<td>Bit-flip, SAF, new value</td>
<td>Changes the data of the register according to the fault mode</td>
</tr>
<tr>
<td>Address decoder</td>
<td>Bit-flip, SAF, new value</td>
<td>Changes the address of the register according to the fault mode</td>
<td></td>
</tr>
<tr>
<td>RAM</td>
<td>Address decoder</td>
<td>Bit-flip, SAF, new value</td>
<td>Changes the data according to the fault mode</td>
</tr>
<tr>
<td>Memory cell or R/W logic</td>
<td>Bit-flip, SAF, static and dynamic faults according to [11]</td>
<td>Changes the address according to the fault mode</td>
<td></td>
</tr>
</tbody>
</table>
TABLE II: Examples of Dynamic Software Diversity Techniques and their Goals in Literature

<table>
<thead>
<tr>
<th>Method</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data re-expression [23]</td>
<td>Security, Reliability</td>
</tr>
<tr>
<td>MEMORY LAYOUT DIVERSITY</td>
<td></td>
</tr>
<tr>
<td>Memory gaps between objects [24]</td>
<td>x</td>
</tr>
<tr>
<td>Changing base address of program/stack [24]</td>
<td>x</td>
</tr>
<tr>
<td>Permutation of the order of variables [24]</td>
<td>x</td>
</tr>
<tr>
<td>Address space layout randomization (ASLR) [25]</td>
<td>x</td>
</tr>
<tr>
<td>INSTRUCTION DIVERSITY</td>
<td></td>
</tr>
<tr>
<td>Insertion of NOP instructions [26]</td>
<td>x</td>
</tr>
<tr>
<td>Instruction loading diversification [27]</td>
<td>x</td>
</tr>
<tr>
<td>Binary stirring [28]</td>
<td>x</td>
</tr>
<tr>
<td>Program encoding randomization [29]</td>
<td>x</td>
</tr>
</tbody>
</table>

IV. ONGOING ACTIVITIES

Currently, we are performing proof-of-concept implementations to evaluate dynamic software diversity technique regarding their effectiveness in detecting common-cause software bugs and hardware faults. More specifically, we analyze diversity techniques that randomize the memory layout of programs. Our future work also includes further evaluations of the proposed fault tolerance techniques. For example, we plan to quantitatively assess the fault tolerance techniques regarding their impact on reliability, availability and safety.

Furthermore, we prepare a open source release of the QEMU-based fault injection framework. Additionally, a demonstration showing that the framework is also beneficial to increase the security of software-based embedded systems is part of ongoing work. We show how to use the framework to evaluate software countermeasures against fault attacks.

Finally, we plan to apply the proposed approaches in an industrial project with the goal to develop next-generation high-reliable controller for hydro-electrical power plants.

REFERENCES