Project Number: FP7-611404

Software impact on System Reliability: Metrics and Models

Authors

A. Savino, G. Di Natale, A. Bosio, M. Kooli, S. Di Carlo, G. Gambardella, D. Gizopoulos, N. Foutris, S.Tselonis, M.Kaliorakis, F. Reichenbach, A. Grasset, T. Loekstad, R. Canal

Version 1.5 – 05/08/2014

Lead contractor: Politecnico di Torino

Contact person:

Alessandro Savino
Control and Computer Engineering Dep.
Politecnico di Torino
C.so Duca degli Abruzzi, 24 I-10129 Torino TO Italy

Tel. +39-011-090.7198

Fax. +39-011-090.7099
E-mail: alessandro.savino@polito.it

Work package: WP4

Affected tasks: T4.2, T5.1

Nature of deliverable[1]	R	P	D	O
Dissemination level[2]	PU	PP	RE	CO

[1] *R: Report, P: Prototype, D: Demonstrator, O: Other*

COPYRIGHT

[2] ***PU**: public, **PP:** Restricted to other program participants (including the commission services), **RE** Restricted to a group specified by the consortium (including the Commission services), **CO** Confidential, only for members of the consortium (Including the Commission services)*

INDEX

Scope of the document

This document is an outcome of task T4.1, "***Software impact on system reliability: Metrics and Models***", elaborated in the description of work (DoW) of the CLERECO project under the Work Package 4 (WP4).

This document aims at describing the *impact of software on system reliability* in terms of metrics and models. With the term *software* we consider here both the system software (e.g., the operating system) and the application software. In particular, this document focuses on modeling the way hardware errors can reach the software stack, and on a preliminary analysis of possible methods to investigate how these errors are propagated and/or masked through the software stack thus affecting the final system reliability. It has to be pointed out that the CLERECO project does not focus on software bugs/errors but only on the effect of hardware faults and their propagation to the software layers.

The document is organized in the following sections:

- **Introduction**. This section sets the background for the document. The objectives of the document and the investigations made for its development.
- **Software Fault Models.** This section describes how hardware faults that are not masked at the hardware level can be modeled at the software level in order to analyze their effect on the software execution.
- **Impact of software on system reliability**: the virtual instruction set defined for the analysis of Software Resilience to be developed in CLERECO. Moreover, this section introduces a preliminary description of the environment that will be implemented, using both already available tools and specific solutions that are going to be developed within the WP.
- **Software faulty behaviors**: this section identifies a set of common faulty behaviors that the software manifests when affected by faults. These software faulty behaviors are organized and analyzed to properly define the Reliability Metrics for Software investigation in CLERECO.

1. Introduction

System reliability has become an important design aspect for computer systems due to the aggressive technology miniaturization, which introduces a large set of different sources of failure for hardware components [1][2][3][4]. Unreliable hardware components affect computing systems at several levels. Raw errors are strongly related to the technology used to build the hardware blocks composing the system and are caused by effects such as physical fabrication defects, aging or degradation (e.g., NBTI), environmental stress (e.g., radiations), etc.

After a raw fault manifests in a given hardware block, it can be propagated through the different hardware structures composing the full system. Even if several faults can be masked during this propagation either at the technology or at architectural level ([4][5][8][43]), some of them can possibly reach the software layer of a system by corrupting either data or instructions composing a software application. These errors can jeopardize the correct software execution producing erroneous results if the computation is completed, or even preventing the execution of the application by causing exceptions, abnormal terminations or leading to an application hang-up. This may have a serious impact on the overall reliability of the system. The software stack itself can play an important role in masking errors generated in the underlying layers. This capability can be further improved by the implementation of software fault tolerance mechanisms [6][7], which enable the improvement of the system reliability but often incur a significant performance overhead. Therefore, the role of the software stack in the overall system reliability is carefully considered in CLERECO.

To avoid misunderstanding with terminology used in different research domains, it is important to clarify in this document that CLERECO focuses on the effect that raw hardware faults reaching the software stack produce on the correctness of the application outcome (usually represented by the result of the software computation). The software stack is seen as a path in which hardware faults can be propagated amplifying and/or masking their effect on the correctness of the expected system's outcome. Software reliability engineering, including software-testing techniques aimed at detecting software design bugs, are out of the scope of CLERECO.

The reliability stack reported in Figure 1 summarizes the basic idea of system reliability evaluation of CLERECO. Every system is split into three main layers: (1) technology, (2) hardware and (3) software. The low-level raw errors of the physical devices are masked in several different ways (addressed in the WP2 of CLERECO Project) as their effect is propagated through the hardware layer (which is evaluated in the WP3) and the software layer of the system stack towards the final program/application outputs. It is the CLERECO's goal to contribute with a full system reliability estimation methodology, which takes into consideration all these factors (technology, hardware and software) to provide an accurate estimate of the expected reliability of the system as early as possible during the design.

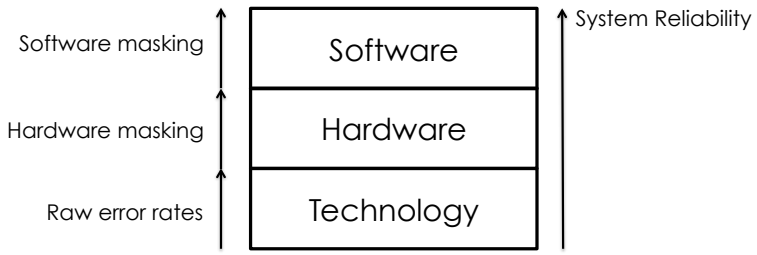

Figure 1: CLERECO Reliability stack

One of the main goals pursued in CLERECO is to be able to analyze the three layers reported in Figure 1 in isolation, and to later combine the outcome of this local analysis in order to infer reliability measures at the system level. This is motivated by the requirement of analyzing very complex systems in which considering all layers at the same time is computationally infeasible.

Each layer included in Figure 1 defines an interface with the upper layer, which in turns sets how the errors can be propagated from one layer to the next one. In this document we focus on errors that can cross the interface between the hardware and the software layer. They are then propagated through the software execution, thus impacting the result of the computation, i.e., the *software outcome*. The portion of the reliability stack considered in this deliverable with the main relevant elements required to analyze the impact of software on the reliability of a system is shown in Figure 2.

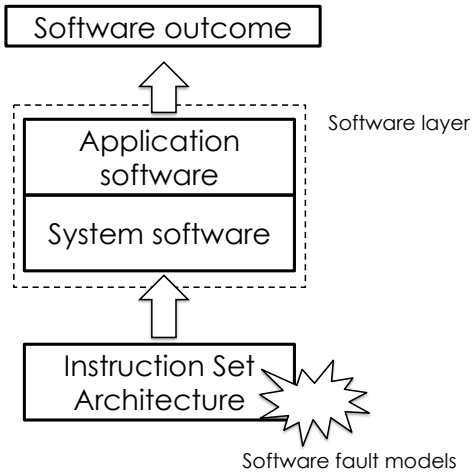

Figure 2: The portion of the system reliability stack considered in this deliverable

The software layer considered in CLERECO includes both the system software (i.e., the operating system) and the application software. Since the global system outcome is commonly represent by the outcome of the software executed in the system (both application and system software), analyzing the software impact on the system reliability implies analyzing the way the software reacts on faults that reach its interface with the hardware layer. In general, the Instruction Set Architecture (ISA) of the target hardware platform executing the software defines this interface between the hardware and the software.

The ability of a software component to mask and/or intrinsically tolerate errors coming from the hardware is also referred in the literature as *software resilience* [24]. This term will therefore

be used within this document and the following deliverables as a synonymous of software masking capability or software impact on the system reliability.

To define the resilience of a software application, it is necessary to evaluate the probability of functional correctness of the software in the presence of hardware faults that propagate in either the software data or software instructions. The next sections will report the effort performed within CLERECO to identify models and metrics to properly represent faults reaching the software layer interface, to analyze how these faults are propagated in the system reliability stack and finally to classify how the software outcome is impacted by these faults.

2. Software Fault Models

Most of the literature that aims at considering the impact of software in the reliability of a full system still starts from low-level hardware faults [31][32][33], trying to propagate them through the hardware architecture to the software layers in order to evaluate their impact on to the final system outcome [34][35][36]. This, in general, requires complex and time-consuming simulations of hardware models that do not enable to analyze complex software stacks. As previously stated, in CLERECO, we aim at detaching the analysis of the software level from the hardware level. We therefore need to model how hardware faults manifest at the software level.

In [34], authors look at the symptoms of transient faults in microprocessors. This observation is performed at a high abstraction level, very close to the ISA level. Other works use simulations to generate fault dictionaries that capture the manifestations from the lower level "off-line" and use them to propagate fault effects during high-level simulations, [37]. Intermittent and Permanent faults started gaining attention very recently. A set of works tries to investigate the effect of propagation of these types of fault up to the software level, [38][39]. However, most of the available works still lack a general abstraction about the effect of hardware faults at the software level.

It is important to highlight here that the main interaction point between the hardware layer and the software layer is the ISA of the microprocessors and co-processors (e.g., accelerators such as GPUs or crypto devices) available in the system. A straightforward way to model the fault propagation from the hardware layer to the software layer is therefore to map hardware faults into a set of fault models that affect the ISA instructions and their data. This somehow detaches the software analysis form the underlying hardware analysis and moves the work of combining the obtained results later on. When the effect of hardware faults to the software layers is accurately modeled at the interface between the hardware and the software (i.e. the ISA and the data), significantly more complex software stack architectures can be studied and the effect of faults at the full system level can be correctly analyzed.

Table 1 provides a preliminary taxonomy of software fault models defined at the ISA level that will be considered in CLERECO. It is worth mentioning here that this taxonomy will be continuously updated during the project to reflect the results provided by WP3 during its analysis of relevant classes of hardware components considered in CLERECO. We explicit do not refer to registers but to (generic) operands in order to maintain a high level description. All considered models apply both to system and application software.

Table 1 - Software Fault Models

Software Fault Model	Description
Wrong Data in an Operand	An operand of the ISA instruction shows an incoherent value (e.g., a value that differs from an expected one).
Not-accessible Operand	An operand of the ISA instruction cannot be addressed to change/retrieve its value.
Operand Forced Switch	An operand is used in place of another, at execution time.
Instruction Replacement	An instruction is used in place of another (either a valid or an invalid one). Transition tables can be provided to guide

Software Fault Model	Description
	the substitution based on statistical evidences in most common ISA.
Faulty Instruction	The instruction produces a wrong result.
Control Flow Error	The flow of control is not respected (control-flow faults).
External Peripheral Communication Error	An input value (from a peripheral) is corrupted or not arriving
Signaling Error	An internal signaling (exception, interrupt, etc.) is wrongly raised or suppressed.
Execution timing Error	An error in the timing management (e.g. PLL) interferes with the correct execution timing.
Synchronization Error	An error in the scheduling processes causes an incoherent synchronization of processes/tasks.

All models reported in Table 1 are very generic and not tight to a specific ISA, so they can be applied to several classes of hardware components. To avoid loosing contact with the underlying hardware layer it is however important to correlate each fault model with candidate hardware fault locations. Table 2 shows a first attempt to perform this mapping that will be constantly improved and refined during the project to support models and algorithms developed within WP5.

Table 2 - Software Fault Model correlation with Hardware Fault Location

Software Fault Model	Location				
	Microprocessor	**Accelerators**	**Memories**	**Peripherals**	**Interconnect**
Wrong Data in Operand	• Register file • Program counter • Buffer • Fetch buffer • Reorder buffer • Load buffer • Store Buffer • Instruction scheduler • Issue queue • ALU • FPU • Pipeline latches	• Register file • Instruction buffer • Score-board • Processing Units	• Main Memory • Cache Memories • TLB	• DMA controller	• Infiniband • Ethernet • Gemini interconnect • Myrinet • Fat Tree • Bi-Directional Link • Wishbone • AMBA • Pair of Northbridge and Southbrige
Not-accessible Operand	• Register File • Fetch buffer • Store Buffer • Issue queue • Reorder buffer	• Register file • Instruction buffer • Operand collector	• Main Memory • Cache Memories	• DMA controller	

Software Fault Model	Location				
	Microprocessor	**Accelerators**	**Memories**	**Peripherals**	**Interconnect**
	• Pipeline latches	• Score-board			
Source Oper-and Forced Switch	• Register File • Instruction decoder • Fetch buffer • Load buffer • Store Buffer • Instruction decoder • Issue Queue • Reorder buff-er • Pipeline latches	• Register file • Instruction buffer • Operand collector	• Main Memory • Cache Memories	• DMA controller	
Instruction Re-placement	• Program Counter • Buffer • Microcode storage • Fetch buffer • Load buffer • Store Buffer • ICache • Instruction decoder • Issue Queue • Reorder buff-er • Pipeline latches	• Instruction buffer	• Main Memory • Cache Memories • TLB	• DMA controller	
Control Flow Error	• Program counter • Microcode storage • Fetch buffer • Reorder Buff-er • Load buffer • Instruction Decoder • Instruction scheduler • Issue queue • Pipeline latches	• Instruction buffer • Handlers of branch diver-gence	• Main Memory • Cache Memories	• PIC	
External Periph-	• Buffer • Microcode		• Main Memory	• DMA controller	• Infiniband • Ethernet

Software Fault Model	Location				
	Microprocessor	**Accelerators**	**Memories**	**Peripherals**	**Interconnect**
eral Communication Error	storage • Pipeline latches		• Cache Memories • TLB • Local/Private Memory	• PIC • UART • PCI Express • USB • Bluetooth	• Gemini interconnect • Myrinet • Fat Tree • Bi-Directional Link • Wishbone • AMBA • Pair of Northbridge and Southbrige
Signaling Error	• Microcode storage • Reorder buffer • Instruction decoder • Instruction scheduler • Pipeline latches	• Operand collector	• Main Memory • Cache Memories • TLB	• DMA controller • UART • PCI Express • USB • Bluetooth	• Infiniband • Ethernet • Gemini interconnect • Myrinet • Fat Tree • Bi-Directional Link • Wishbone • AMBA • Pair of Northbridge and Southbrige
Execution timing Error	• Program counter • Branch predictors • Branch target buffers • Return Address Stack • Fetch buffer • Reorder buffer • Instruction decoder • Instruction scheduler • Issue queue • Pipeline latches	• Operand collector • Scheduler of blocks or work groups	• Main Memory	• DMA controller • PIC • UART • PCI Express • USB • Bluetooth	
Synchronization Error	• Instruction scheduler • Pipeline	• Score-board • Operand	• TLB	• DMA controller • PIC	

Software Fault Model	Location				
	Microprocessor	**Accelerators**	**Memories**	**Peripherals**	**Interconnect**
	latches	collector • Scheduler of blocks or work groups • Vector Processing Unit		• UART • PCI Express • USB • Bluetooth	

3. Impact of software on system reliability

While the definition of methods to analyze the impact of software on system reliability are out of scope of this deliverable and will be properly described in deliverables D4.2.1, D4.2.2 "**Software Characterization Methods**" that will be released later in the project (M12), a preliminary analysis and discussion is required to proper setup the background for the core activity of WP4.

Similarly to the hardware layer, the software layer has several error masking/amplifying capabilities. Figure 3 summarizes the main error masking effects at software level [7].

An error can be masked at the operating system level if:

- It affects the architecture state components that are not used by the OS.
- It does not raise any Fatal Trap or Hang.
- It does not affect the process state of the current application.

An error can be amplified at the operating system level if:

- It affects hardware status resources (e.g., machine control registers, operating system control registers).
- It affects the operating system memory management module.
- It affects the processes/threads scheduling capabilities.
- It propagates to any other resource.

An error can be masked by the application software if:

- It affects the control flow without leading to an abnormal application exit or skip of functions.
- It affects the program without leading to an output that differs from the expected one at the expected time.
- It does not raise any Fatal Trap or Hang of the application.
- It has no impact on the execution times and on the responsiveness of the application.

An error can be amplified by the application software if:

- It affects the control flow leading to wrong paths.
- It affects application resources in their initialization phase.
- It raises wrong Fatal Traps.
- It propagates to any other resource.

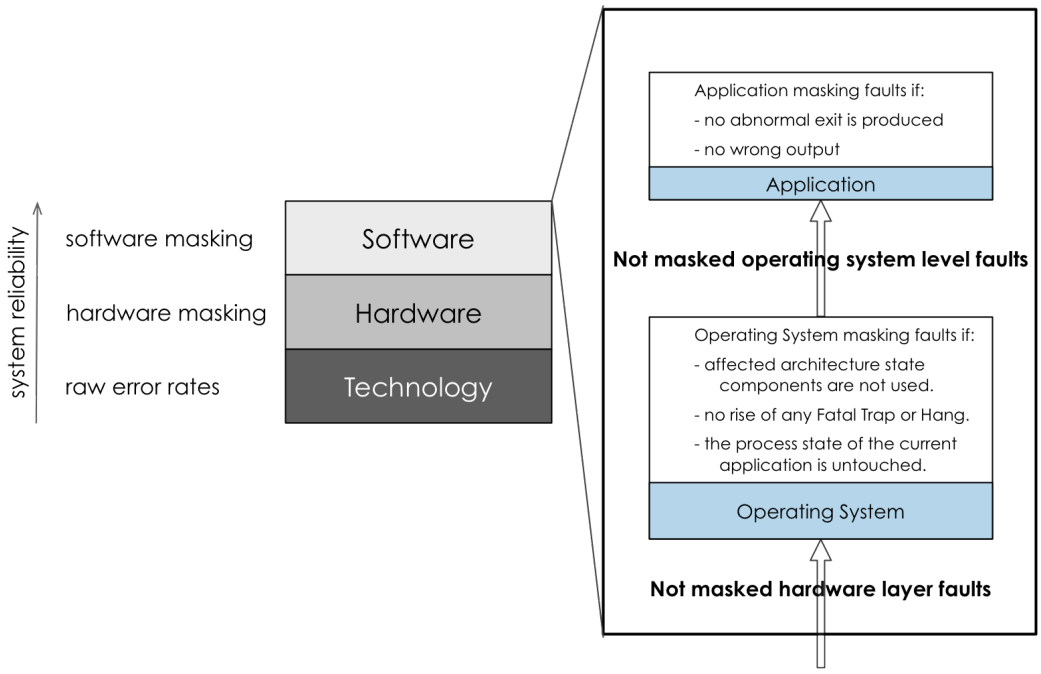

Figure 3: Masking effects on System Vulnerability Stack

Analyzing this masking capability requires to set a common ground, in terms of execution platform, to cope with the huge diversity of available software platforms and computer architectures.

Common software engineering techniques resort to high-level programming languages (e.g., C/C++ or Java) or data-flow (graphical) programming languages (e.g., MathWork Simulink) to describe software routines independently from the target execution platform[3].

High-level programming languages are then mapped to the ISA of the final system. The mapping can be done statically (at compile time) or dynamically (at run-time). In both cases the high-level program is translated into a sequence of low-level instructions that can be executed by the selected hardware thus creating a very specific link between the software layer and the hardware layer of a system (Figure 1).

This is in contrast with the CLERECO main idea, which aims at performing reliability evaluation in the early stage of the system design when the selection of the hardware is still an open choice. This in turns requires investigating methods and tools to model the software independently from the target hardware architecture, and to link later on the results of the software analysis to the specific reliability metrics collected for the selected hardware architecture. At this early stage the target ISA is still unknown and therefore cannot be exploited either to define fault models according to the taxonomy presented in Table 1 or to perform simulations to analyze how faults propagate through the software modules.

[3] *Embedded systems applications, which highly depend on the system platform and/or peripheral drivers in common operating systems, may represent an exception to this practice.*

In this scenario, one of the natural solutions to tackle this problem is to resort to virtualization techniques enabling to abstract the ISA used to describe the software from the target hardware architecture. This solution introduces an additional abstraction layer between the hardware ISA and the software layer that further decouple the hardware and the software layers [16].

The concept of software virtualization is gaining increasing importance since it ensures efficient and flexible performance, and enables cost saving from sharing the same physical hardware. Therefore, several projects proposing virtualization infrastructures are available. The next subsections report the effort performed in CLERECO to identify a candidate virtualization framework that can serve as a starting point to implement the infrastructure required to analyze parameters that are relevant to evaluate the software resilience to hardware faults.

3.1. Abstract Instruction Set Architecture

Virtualization technologies can separate hardware and software management and provide useful features, including performance isolation [46]. Moreover, virtualization technologies can also provide portable environments for the modern computing systems [47]. In fact, a virtual machine (VM) is a logical machine having almost the same architecture of a real host machine, running an operating system in it. Virtual machines allow users to create, copy, save (checkpoint), read and modify, share, migrate and roll back the execution state of a machine with simple file manipulation tools. This flexibility provides significant value for users and administrators. Traditionally, virtual machines have focused on fairly sharing the processor resources among domains, [47].

In the WP4, one of the main aims is to investigate the software reaction to hardware faults, without knowing the target hardware architecture. In this context, a virtualized environment perfectly matches our needs

Following [16], available technologies for the implementation of VMs can be classified in two main categories:

- *System Virtual Machines* (VMs) provide a complete environment that supports the execution of a complete operating system. System VMs issue a platform to run programs in which the real hardware is not available for use, and to run multiple OS environments concurrently on the same computer with a strong isolation. The virtual machine relies on an ISA that is different from the one of the physical machine. In this situation the whole software is virtualized, therefore the VM has to emulate both the application and the OS code.
- *Process Virtual Machines* are virtual platforms that execute a single process. The VM is created when the process is started and deleted when it terminates. Its goal is to provide an independent programming environment platform, which abstracts the details of the underlying hardware or operating system, and enables a program to execute in the same way on any platform. Process VMs using different guest and host ISAs are implemented using an interpreter, which fetches, decodes and emulates the execution of individual guest instructions. Since this process is relatively slow, dynamic binary translation can provide better performance by converting guest instructions to host instructions in blocks rather than instruction by instruction, and saving them in a cache for later reuse.

The main drawback of most available System and Process VMs is that they are still bounded to specific ISAs implemented by real hardware architectures. In CLERECO we need to overcome this limitation by working with an ISA that is independent from the final hardware used in the system. As depicted in Figure 4, we therefore plan to identify a virtualization environment

able to develop software using a Virtual Instruction Set Architecture (VISA), creating an additional layer between the software stack and the actual hardware architecture. The VISA must be later translatable into a real ISA in order to better analyze the software behavior on the target system when the hardware architecture is finally defined.

Both the Operating System and the Application Software will be described in CLERECO according to this VISA. Following [42], adding this additional level of abstraction still enables us to investigate the error propagation properties of the software and to efficiently correlate them with errors arising in the actual hardware.

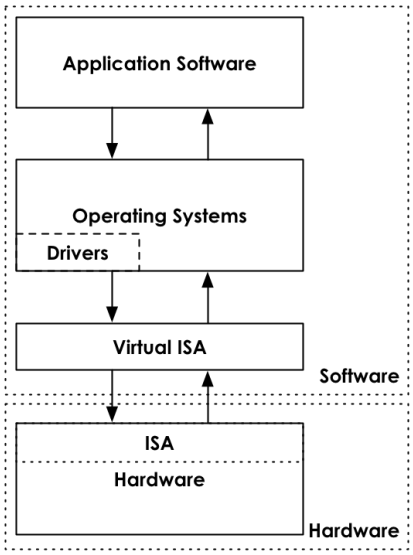

Figure 4: CLERECO System Stack exploited in terms of the Software execution stack

There are several frameworks in the literature to use virtualization with virtual instruction sets to perform complex analysis of software applications on different architectures [19][25][26][27][28]. The software under analysis is compiled into a sequence of *abstract* instructions. Software compiled using this abstract instruction set can then be directly executed on a specific host microprocessor's architecture (via further translation/synthesis, [25][26]) or on virtual machines (without requiring further translations).

WP4 is investigating different alternatives of available virtualization environment implementing VISAs to exploit for the analysis of software resilience. Three realistic options have been considered so far:

- Java. Java applications are typically compiled to byte-code (class file) that can run on any Java virtual machine (JVM) regardless the underlying microprocessor architecture. The Java byte-code is a form of ISA virtualization. Based on Java, Jaca is a fault injection tool that is able to inject faults in object-oriented systems and can be adapted to any Java application without need of its source code. To perform injection it is enough to know just few information about the application like the classes, methods, and attributes names [44].
- .NET / Mono. The .NET framework follows the same philosophy of Java. It consists of a virtual machine able to run Common Languages Infrastructure (CLI) code. The CLI is an object-oriented VISA that is the lowest level of the framework, [49]. Mono is the open version of the .NET VM, [50]. To the best of our knowledge, no fault injection environment is actually available.

- LLVM. LLVM is a Process Virtual Machine that implements a virtualized instruction set architecture. A wide community of developers (including Intel [29], NVIDIA[30], and others) uses and develops tools having LLVM at their core. Among them, several fault injection tools are currently developed [23],[24],[45]. They are research tools released under the open source code licenses.

Among the considered environments a very promising solution for CLERECO is LLVM. Java, while being widely used in web-based applications, has the disadvantage of not being really suitable for both HPC and embedded applications, where the final application will be designed specifically for the target hardware in order to fully exploit all capabilities (speed, power) of the system. Moreover, the JVM is restricted to the Java programming language thus limiting the spectrum of software that can be analyzed.

While LLVM is not the final decision for the implementation of the CLERECO software analysis framework, and further investigations will be performed, in the next subsections we will report additional information regarding the LLVM infrastructure in order to highlight interesting features and missing functionalities of this environment.

3.1.1. The LLVM Project

LLVM (formerly Low Level Virtual Machine) [17] is a compiler infrastructure designed for compile-time, link-time, run-time optimization of programs written in arbitrary programming languages (see Figure 5). Originally implemented for C and C++, nowadays, the languages with compilers that use LLVM include D, Fortran, Julia, Objective-C, Python, Ruby, Rust, Scala, C# and so on. It also supports, as back ends, a huge set of ISAs: ARM, MicroBlaze, MIPS, NVidia PTX (called "NVPTX' in LLVM documentation), PowerPC, SPARC, x86/x86-64, and so on. Moreover, modern programming paradigms and architectures, such as GPU accelerators and Intel Phi architectures, are supported both in terms of front-end (i.e., CUDA C/C++ API) and backend, [10],[11].

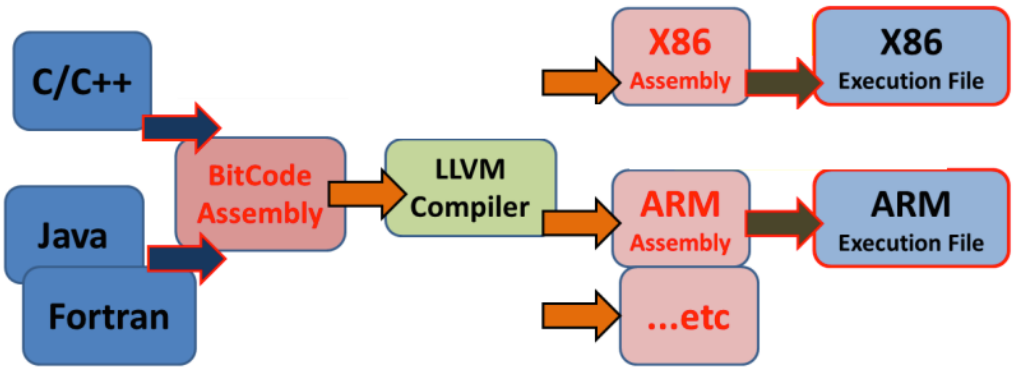

Figure 5: LLVM Abstraction Example

Together with the full tool chain required for software design (e.g., compiler, optimizer, etc.), LLVM provides a set of additional tools explicitly devoted to perform investigation of different software properties. The LLVM tool kit includes:

- **The LLVM Core**: it includes a code optimizer and the generator.
- **CLang**: it is a native C/C++/Objective-C compiler.
- **Dragonegg**: a tool for the integration with GCC parsers.
- **LLDB**: a native debugger.
- **Libc++**: a native C++ Standard Library implementation.

- **Compiler-rt**: one of the most useful tools for the project. It is a low-level target specific code generator, which includes a set of promising facilities:
    - **Sanitizers' runtimes**: runtime libraries to run the code with sanitizer instrumentation. This includes a **Data Flow Sanitizer** to perform a dynamic flow analysis and an **Address Sanitizer** to help in the memory error detection.
    - **Profile**: a library to collect profiling information about the software.
- **OpenMP**: a native OpenMP implementation, particularly helpful for implementing parallel version of single-threaded algorithms.
- **vmkit**: a native JAVA and .NET virtual machine.
- **Klee**: is a symbolic virtual machine that uses a theorem prover to evaluate the dynamic paths of software. It has been successfully used for software testing purposes [21].

LLVM is also supported by a large community of developers, which contributes with extra components. The open source license, the continuous updates, the high availability of tools, and the large community of users make LLVM a good starting point for the CLERECO software analysis infrastructure.

Eventually, as it will be described in the next subsection, the LLVM project comes with a well-known implementation of simulation environments, which also include fault injection/simulation engines.

3.1.2. The LLVM Simulation Environment

This subsection aims at summarizing the LLVM state-of-the-art tools to build a simulation environment able to collect data about software resilience at high level.

3.1.3. LLFI

LLFI [23],[45] is an LLVM based fault injection tool that enables to inject faults into the LLVM intermediate level of the application source code. Using LLFI, faults can be injected at specific program points and data types. The effect can be easily tracked back to the source code. LLFI is typically used to map fault characteristics back to source code, and to understand program characteristics or source level for various kinds of fault outcomes. The reason why LLFI injects faults at this level is that the LLVM intermediate code is at a higher level than the assembly code, and is able to encode more information than the source code. In fact, at the assembly level, it is not easy to track back the fault behavior to the source level. This problem could be solved with a fault injection at the source code level. However, this solution does not allow modeling hardware faults because many hardware faults, that affect some control flow instructions and registers are masked at the lower layers of the system and cannot be simulated at the application layer.

The goal behind LLFI is to identify source level heuristics that enable to identify optimal locations for high coverage detectors of faults causing Egregious Data Corruptions (EDCs). EDCs are application outcomes that deviate significantly from the error-free outcome [6]. Non-EDCs are application outcomes with small deviations in output. EDCs and non-EDCs define the Silent Data Corruptions (SDCs), which are the outcomes that result from any deviation from the fault free outcome. A threshold between EDC and non-EDC can be defined; if set to zero, even single bit errors are considered SDCs.

LLFI supports fault injection errors that model the effect of transient hardware faults occurring in the processor (e.g., errors caused by cosmic ray or alpha particle strikes affecting flip flops and logic elements). It considers faults in the functional unit (the ALU and the address computation for loads and store). However, the tool does not consider faults in the memory

components, in the control logic of the processor, and in the instructions, which is a huge limitation of the approach.

3.1.4. KULFI

KULFI [24] (Kontrollable Utah LLVM Fault Injector), developed by Gauss Research Group at School of Computing, University of Utah, Salt Lake City, USA, is an LLVM based fault injection tool, which enables to inject random single bit errors at instruction level. It allows injecting faults into both data and address registers. It simulates faults occurring within CPU state elements, providing a finer control over fault injection. For example, it enables the user to define options related to the fault injection mechanism, such as the probability of the fault occurrence, the byte position in which the fault could be injected, the suitable choice whether the fault should be injected into the pointer register or the data register.

KULFI considers the injection of both dynamic and static faults. Dynamic faults represent transient faults and they are injected to a fault site randomly selected during program execution. Static faults represent permanent faults and are injected to a fault site selected randomly before the program execution.

3.2. Software Development Scenarios

In addition to the classical C/C++ development scenario, we identified two de-facto standards for development environments exploited in real software contexts. The two environments have been identified to support the LLVM introduction as Virtual ISA. They are deeply investigated since we plan to resort to them for the demonstrator implementation expected in WP 6.

3.2.1. Simulink

The first scenario we are going to describe in terms of LLVM impact is a common early stage software development: modeling via Simulink. It is common to start developing the software by designing its flow via Simulink modules. The main advantage is the creation of an interactive model, which can be already used for simulations and test cases.

LLVM does not natively supports to compile and execute Simulink models. Nevertheless, MathWorks provides a set of add-ons to the Simulink development environment (e.g., the Simulink Coder [13] and the Embedded Coder [12]), able to generate C and C++ code from Simulink diagrams and Stateflow charts that can be then compiled for the execution within LLVM using the LLVM Clang Compiler [9] as reported in Figure 6.

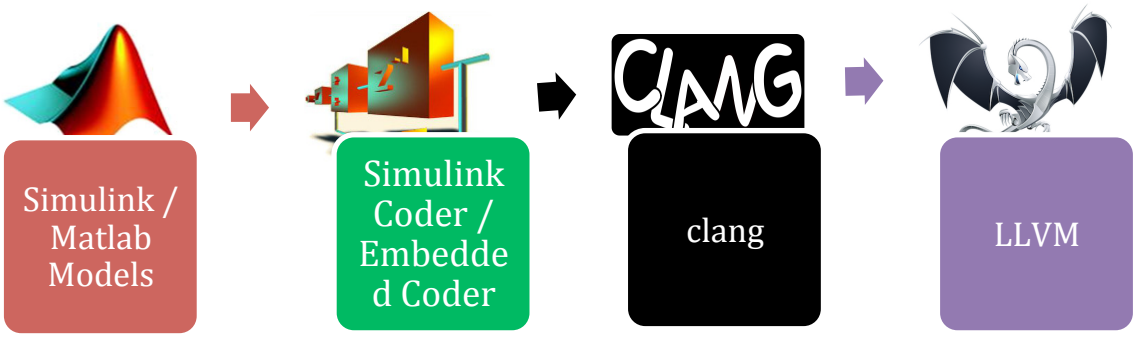

Figure 6: Simulink integration with LLVM

In details, the Simulink Coder tool allows the user:

- ANSI/ISO C and C++ code for discrete, continuous, or hybrid Simulink and State-flow models generation;
- Incremental code generation;
- Integer, floating-point, and fixed-point data type support;
- Code Generation for single-rate, multi-rate and asynchronous models;
- Single-tasking, multitasking, and bare-board compatibility;
- External mode simulation for code tuning and monitoring.

It therefore provides a very general way to transform Simulink models into programs that can be executed and analyzed in LLVM.

Another feasible alternative is to exploit the MathWorks Embedded Coder, an add-on designed for the generation of C code for embedded systems. The tool can be set-up in order to optimize the code for specific target architectures (e.g., ARM, AMD, Freescale, Intel, TI), thus extending the native capabilities of Simulink Coder. For more information please refer to [13]. Differently from the Simulink Coder, this translation performs a preliminary assumption on the target microprocessor architecture. It cannot therefore be used in the very early stages of the design process, but only in the later stages when the target hardware platform has been identified. Moreover, this tool can be exploited to generate different versions of the software that can be compared in the study of the overall system reliability.

It is important to notice that there are several limitations on the Simulink blocks accepted as part of the input model. In fact, the coder is not able to generate the corresponding C code for all available Simulink blocks. Industrial partners may investigate their usage of Simulink models to fit the issue. For more information on the limitations refer to [14].

3.2.2. SCADE Suite

The second development scenario considered here is a development based on a formal description in the SCADE Suite. The SCADE Suite is an integrated design environment and development for critical embedded software applications. The tools enable graphic design, verification through simulation and formal methods, and certified code generation. Products requirements management, configuration management and automatic documentation generation are also included, reducing the time of certification applications. The tool suite is dedicated to the development of critical embedded applications using formal description, in industries Aerospace, Defense, Rail Transport, Energy and Industry.

Similarly to the Simulink scenario, LLVM is not integrated natively within the SCADE Suite, but it can be used for the compilation of the code generated by the tool suite (see Figure 7).

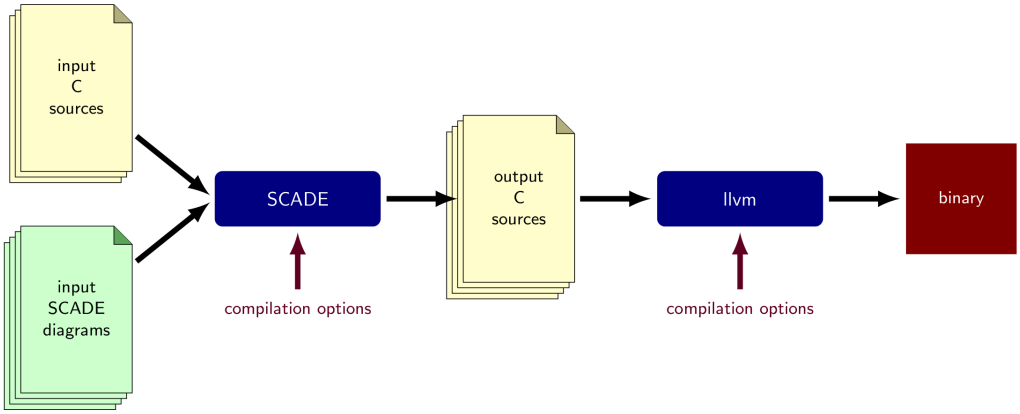

Figure 7: SCADE Suite integration with LLVM

The SCADE Suite KCG code generator is certified/qualified according to following international safety standards:

- DO-178B qualified up to Level A and DO-178C Ready for Civilian and Military Aeronautics.
- IEC 61508 certified at SIL 3 by TÜV for Industry.
- EN 50128 certified at SIL 3/4 by TÜV for Rail Transportation.
- IEC 60880 compliant for Nuclear Energy.
- ISO 26262 for Automotive.

The SCADE Suite generates ANSI C code that can be directly compiled with LLVM framework. LLVM can thus fit in different development scenarios used in different application domains.

4. Software Faulty Behavior Classes

Once the hardware faults that can affect the software layer have been properly modeled, and a platform to analyze how these faults propagate during the execution has been identified, the investigation of the impact of these faults on the software still requires analyzing how they affect the final software result. Basically, two main classes of errors can be observed at the output interface of the software layer [18]:

1. **Incorrect output:** the hardware error propagation only affects the output of the application.
2. **Application failure:** an error produces a software failure that forces the application to interrupt its execution prematurely by crashing or keeping it in an unresponsive state (e.g., hangs). In this case, no output might be produced.

In both situations, the key concept is that, whenever the error is located in software, there exists an error masking chain that links it to the underlying reliability layer. Nevertheless, this very simple classification requires to be elaborated in order to identify a more fine-grained set of behaviors, named here *software faulty behaviors (SBFs),* the software may produce when affected by the faults defined in Section 2.

In a complex system, the SBF combines both the behavior of the operating system activity and the application software activity. Some publications try to define possible classes of SBFs to be used for the analysis of Software Resilience.

By the analysis of the literature [15][20][22][40][34][41], three main classes of SBFs can be defined:

1. **Timing**: they are related to the ability of the software to respect the target time constraints. Since the time is always related to a metric, a set of *metrics* (e.g., IPC, clock cycles, etc.) will be proposed and discussed, in order to deal with the accuracy of the timing in proper ways.
2. **Unresponsiveness**: once the software is affected by a fault, it may become unresponsive, thus (most of the time) preventing the production of its outputs. *Full unresponsiveness* arises when the whole software stack crashes (both the application and the OS). *Partial unresponsiveness* is instead generated whenever the operating systems remains active, while the application software stops working. Whenever the unresponsiveness can be detected only by analyzing some time-related characteristic of the system execution, a set of proper *metrics* and related *parameters* have to be defined to cope with that.
3. **Data Integrity**: this class comprises errors in application data. Two cases may arise: (1) the result produced by the application is correct (Benign) and no faulty behavior is observed; (2) the output contains errors (Data Corruption, DC). DCs may be further split into two sub cases. Applications that provide an output that meaningfully differs from the expected one lead to Egregious Data Corruption (EDC), while applications that generate *small* changes (where the meaning of *small* is discretional) produce so called Non-Egregious Data Corruption (Non-EDC). This latter case is mainly a matter of how the System Requirements describe the ability of the application to tolerate errors in the final output.

Table 3 summarizes a preliminary set of possible SBFs identified in CLERECO. These outcomes can be mapped into the three general classes introduced previously in this section.

Table 3: CLERECO Software Outcomes

SBF Class	SBF	SBF Subclass	Description
Timing	In-Time		The software execution timing is well respected.
	Wrong timing / Out of sync	Early	Given the expected execution time, the software finishes earlier than expected (due to a **threshold parameter** to be defined). This could also mean that it is out of sync, with respect to the whole system.
		Late	Given the expected execution time, the software ends later than expected (due to a **threshold parameter** to be defined). This could also mean that it is out of sync, with respect to the whole system.
Unresponsiveness	Full Unresponsive	Fatal Hardware Traps [22]	A fatal hardware trap occurs in either the application or the operating system. A fatal trap is typically not thrown during a correct program and can cause the system to shut down.
		Hangs [22]	Hangs due to an abnormal behavior. Usually detected by looking at the all executed branch.
	Partial Unresponsive	Abnormal Application Exit [22]	In a whole system, when an application crashes, the OS is aware of this event, so the system is not unresponsive. Avoiding Abnormal application exit could improve reliability of whole system.
		High OS Activity [22]	The OS is invoked via system trap and the execution remains in the OS without returning to the application.
	Responsive		The system is working as expected. In these cases, the error has been masked by either an internal masking effect in the system or by a protection mechanism added in the design phase.
Data Integrity	Benign		The software only produces correct results (by results we mean mes-

			sages, direct actions to the outside world, etc.). If incorrect results are generated as a consequence of a fault then the software does not output them (remains silent).
	Silent Data Corruptions (SDCs)	Egregious Data Corruption (EDC)	The application outcomes deviate significantly from the fault free outcome. The deviation ratio has to be defined, case-by-case, choosing a **threshold parameter** and the **metric** used to measure that deviation.
		Non Egregious Data Corruption (Non-EDC)	SDCs that result in any deviation in the output from the fault free outcome with small deviation in output that could be tolerate by the system. The deviation refers to the same defined for EDC.

Every time designers need to analyze a given software application, they can define their own custom SBFs as combinations of the ones presented in. Table 3

As an example, for a given application designer may consider as an abnormal situation every case in which the data integrity is corrupted by a SDC, or the system is unresponsive (either fully or partially), ignoring for example timing issues generated by errors (i.e., out of sync execution of the software is not considered as an error).

This preliminary taxonomy, even if simple, should be enough to model most of real situations that may arise during the software execution in presence of faults.

5. Conclusions

Work Package 4 (WP4) aims at analyzing the software resilience to hardware faults considering both system and application software. This report represents the first step toward this goal. It contributes to providing:

- A preliminary set of models to represent hardware faults at the software level.
- The basis to identify a software virtual architecture to analyze software modules.
- A definition for a set of preliminary software fault behaviors to evaluate the effect of the faults on the software results.

The content of this deliverable impact on the way WP4 is going to characterize the Software (refer to Deliverable D4.2.x) modules in the forthcoming future. WP5 will also be affected because it is going to be influenced by software fault models and SBFs when dealing to the input parameters for the estimation model (refer to Deliverable 5.1.x). Moreover, the SBFs help the definition of (new) system level metrics (refer to WP2's Deliverable 2.4.x) and, consequently, on the way the WP5 system reliability estimation model will provide the output of the estimation (refer to Deliverable 5.2.x and 5.3).

6. Bibliography

[1] Robert Baumann, "Soft Errors in Advanced Computer Systems," IEEE Design & Test of Computers, vol. 22, no. 3, pp. 258-266, May/June, 2005.

[2] S. Borkar et al., "Design and Reliability Challenges in Nanometer Technologies", IEEE DAC, pp. 75-75, 2004.

[3] P. Shivakumar, M. Kistler, "Modeling the effect of technology trends on the soft error rate of combinational logic". IEEE DSN, 2002.

[4] S. S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt, T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor", MICRO, pp. 29-40, 2003

[5] D. Ernst et al., "Razor: circuit-level correction of timing errors for low-power operation," IEEE MICRO, Vol. 24, no. 3, pp. 10-20, 2004.

[6] M. Dimitrov and H. Zhou, "Unified Architectural Support for Soft-Error Protection or Software Bug Detection", International Conference on Parallel Architectures and Compilation Techniques, 2007

[7] Nithin Nakka, Giacinto Paolo Saggese, Zbigniew Kalbarczyk, Ravishankar K. Iyer, An architectural framework for detecting process hangs/crashes, Proceedings of the 5th European conference on Dependable Computing, April 20-22, 2005, Budapest, Hungary, doi:10.1007/11408901_8

[8] R. Vadlamani et al., "Multicore soft error rate stabilization using adaptive dual modular redundancy", IEEE DATE, pp. 27-32, 2010.

[9] Clang: a C Language Family frontend for LLVM. http://clang.llvm.org

[10] CUDA LLVM Compiler, https://developer.nvidia.com/cuda-llvm-compiler

[11] Intel ispc compiler, http://ispc.github.io

[12] MathWorks. "Embedded Coder - Datasheet." http://www.mathworks.it/products/datasheets/pdf/embedded-coder.pdf

[13] "Simulink Coder - Datasheet." http://www.mathworks.it/products/datasheets/pdf/simulink-coder.pdf

[14] Simulink Coder - Product Limitations Summary: http://www.mathworks.com/help/toolbox/rtw/ref/brl3tbg.html

[15] Slot, D.T., N.A. Speirs, Z. Kalbarczyk, S. Bagchi, J. Xu, and R.K. Iyer. "Comparing Fail-Silence Provided by Process Duplication versus Internal Error Detection for DHCHP Server." 15th Internetional Parallel and Distributed Processing Symposium. San Francisco, CA : IEEE, 2001.

[16] Smith, J.E.; Nair, R., "The architecture of virtual machines," Computer , vol.38, no.5, pp.32,38, May 2005, doi: 10.1109/MC.2005.173

[17] The LLVM Compiler Infrastructure. http://llvm.org

[18] Rehman, Semeen; Shafique, Muhammad; Aceituno, Pau Vilimelis; Kriebel, Florian; Chen, Jian-Jia; Henkel, Jorg, "Leveraging variable function resilience for selective software reliability on unreliable hardware," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013 , vol., no., pp.1759,1764, 18-22 March 2013, doi: 10.7873/DATE.2013.354

[19] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke, "LLVA: A Low-level Virtual Instruction Set Architecture", Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36), San Diego, California, Dec. 2003.

[20] Thomas, Anna, and Karthik Pattabiraman. "LLFI: An Intermediate Code Level Fault Injector For Soft Computing Applications." 9th Workshop on Silicon Errors in Logic (SELSE-09). IEEE, 2013

[21] Cristian Cadar, Daniel Dunbar, Dawson Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs", USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), San Diego, CA, December 2008

[22] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design", In Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS XIII), 2008. ACM, New York, NY, USA, 265-276. DOI=10.1145/1346281.1346315

[23] A. Thomas and K. Pattabiraman, "LLFI: An intermediate code level fault injector for soft computing applications," 2013.

[24] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan, "Towards formal approaches to system resilience," in Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), 2013.

[25] Balboni, A.; Fornaciari, W.; Sciuto, D.; Vincenzi, M., "The use of a virtual instruction set for the software synthesis of Hw/Sw embedded systems," System Synthesis, 1996. Proceedings., 9th International Symposium on , vol., no., pp.77,82, 6-8 Nov 1996, doi: 10.1109/ISSS.1996.565883

[26] Smith, J.E.; Sastry, S.; Heil, T.; Bezenek, T.M., "Achieving high performance via co-designed virtual machines," Innovative Architecture for Future Generation High-Performance Processors and Systems, 1998, vol., no., pp.77,84, 24-24 Oct. 1998, doi: 10.1109/IWIA.1998.779076

[27] Adve, V.; Brukman, M.; Evlogimenos, A.; Gaeke, B., "Software implications of virtual instruction set computers," Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, vol., no., pp.201,, 26-30 April 2004, doi: 10.1109/IPDPS.2004.1303226

[28] Gremzow, C., "Quantitative global dataflow analysis on virtual instruction set simulators for hardware/software co-design," Computer Design, 2008. ICCD 2008. IEEE International Conference on, vol., no., pp.377,383, 12-15 Oct. 2008, doi: 10.1109/ICCD.2008.4751888

[29] Intel Corporation, "Why we chose LLVM", https://software.intel.com/en-us/blogs/2009/05/27/why-we-chose-llvm

[30] NVIDIA, CUDA compiler, https://developer.nvidia.com/cuda-llvm-compiler

[31] M.Abramovici, M.A.Breuer, A.D.Friedman, Digital Systems Testing and Testable Design, Wiley-IEEE Press, 1994.

[32] M.Bushnell, V.Agrawal, Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits, Kluwer academic publishers, 2002.

[33] S.Mukherjee, Architecture Design for Soft Errors, Morgan Kaufmann, 2008.

[34] N. Wang and S. Patel. ReStore: Symptom-Based Soft Error Detection in Microprocessors. IEEE Transactions on Dependable and Secure Computing, 3(3), July-Sept 2006.

[35] H. Cha et al. A Gate-Level Simulation Environment for Alpha-Particle-Induced Transient Faults. IEEE Transactions on Computers, 45(11), 1996.

[36] S. Mirkhani, M. Lavasani, and Z. Navabi. Hierarchical Fault Simulation Using Behavioral and Gate Level Hardware Models. In 11th Asian Test Symposium, 2002.

[37] Z. Kalbarczyk et al. Hierarchical Simulation Approach to Accurate Fault Modeling for System Dependability Evaluation. IEEE Transactions on Software Engineering, 25(5), 1999.

[38] Rashid, L.; Pattabiraman, K.; Gopalakrishnan, S., "Towards understanding the effects of intermittent hardware faults on programs," *Dependable Systems and Networks Workshops*

(DSN-W), 2010 International Conference on , vol., no., pp.101,106, June 28 2010-July 1 2010 doi: 10.1109/DSNW.2010.5542613

[39] Sharma, A.; Sloan, J.; Wanner, L.F.; Elmalaki, S.H.; Srivastava, M.B.; Gupta, P., "Towards analyzing and improving robustness of software applications to intermittent and permanent faults in hardware," *Computer Design (ICCD), 2013 IEEE 31st International Conference on* , vol., no., pp.435,438, 6-9 Oct. 2013, doi: 10.1109/ICCD.2013.6657076

[40] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults. SIGPLAN Not. 47, 4 (March 2012), 123-134. DOI=10.1145/2248487.2150990 http://doi.acm.org/10.1145/2248487.2150990

[41] Jiesheng Wei; Rashid, L.; Pattabiraman, K.; Gopalakrishnan, S., "Comparing the effects of intermittent and transient hardware faults on programs," *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, vol., no., pp.53,58, 27-30 June 2011, doi: 10.1109/DSNW.2011.5958835

[42] Garcia, P.; Gomes, T.; Salgado, F.; Cardoso, P.; Cabral, J.; Ekpanyapong, M., "Reliability correlation between physical and virtual cores at the ISA level," Emerging Technologies & Factory Automation (ETFA), 2012 IEEE 17th Conference on , vol., no., pp.1,4, 17-21 Sept. 2012, doi: 10.1109/ETFA.2012.6489725

[43] Kaliorakis, M.; Tselonis, S.; Foutris, N.; Gizopoulos, D., "D3.1 – Report on major classes of hardware component"

[44] R. de Oliveira Moraes and E. Martins, "Jaca - a software fault injection tool," Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on, p. 667, June 2003.

[45] Jiesheng Wei, Anna Thomas, Guanpeng Li and Karthik Pattabiraman, "Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults", IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2014.

[46] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. "Enforcing Performance Isolation across Virtual Machines in Xen". In Proceedings of the 7th International Middleware Conference, LNCS Press, 2006. pp.342-362

[47] Yunfa Li; Wanqing Li; Congfeng Jiang, "A Survey of Virtual Machine System: Current Technology and Future Trends," Electronic Commerce and Security (ISECS), 2010 Third International Symposium on , vol., no., pp.332,336, 29-31 July 2010, doi: 10.1109/ISECS.2010.80

[48] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. "Safe Hardware Access with the Xen Virtual Machine Monitor". Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS), Boston, MA, October 2004.

[49] Microsoft Corporation, .Net Framework 4, http://msdn.microsoft.com/en-us/library/vstudio/w0x726c2%28v=vs.100%29.aspx

[50] Mono Project, http://www.mono-project.com