1st International Workshop on Resiliency in Embedded Electronic Systems

Mövenpick Hotel Amsterdam City Centre
Amsterdam, The Netherlands
Octiber 8th, 2015

Final Proceedings

General Chairs
Daniel Müller-Gritschneder, Technical University of Munich, Germany
Wolfgang Müller, Heinz Nixdorf Institute, Germany
Subhasish Mitra, Stanford University CA, USA

October, 2015

Content

CLEAR: Cross-Layer Exploration for Architecting Resilience
Eric Cheng, Lukasz G. Szafaryn, Shahrzad Mirkhani, Hyungmin Cho, Chen-Yong Cher, Kevin Skadron, Mircea Stan, Klas Lilja, Jacob A. Abraham, Pradip Bose, Subhasish Mitra 1

Cross-layer Resilience Mechanisms to Protect the Communication Path in Embedded Systems
Tobias Stumpf, Hermann Härtig, Eberle, A. Rambo, Rolf Ernst 3

Reliability-Aware Task Mapping on Many-Cores with Performance Heterogeneity
Kuan-Hsun Chen, Jian-Jia Chen, Florian Kriebel, Semeen Rehman, Muhammad Shafique, Jörg Henkel 10

Providing Flexible and Reliable on-Chip Network Communication with Real-Time Constraints
Eberle A. Rambo, Rolf Ernst 12

Reliability and Thermal Challenges in 3D Integrated Embedded Systems
Christian Weis, Matthias Jung, Norbert Wehn 15

Improving Code Generation for Software-Based Error Detection
Norman A. Rink, Jeronimo Castrillon 16

Resilient System Design through Symbolic Simulation and Online Diagnostics Methods
Thiyagarajan Purusothaman, Carna Radojicic, Christoph Grimm 21

Checkpointing Virtualized Mixed-Critical Embedded Systems
Mihalis Psarakis, Aitzan Sari 23

Methods of Timing Reliability Improvement for Combinational Blocks in Microelectronics Systems
Sergey Gavrilov, Galina Ivanova 28

Bridging the Gap between Probabilistic Safety Analysis and Fault Injection in Virtual Prototypes
Moomen Chaari, Bogdan-Andrei Tabacaru, Wolfgang Ecker, Cristiano Novello, Thomas Kruse 34

Using Virtual Platform for Reliability and Robustness Analysis of HW/SW Embedded Systems
Reda Nouacer, Manel Djemal, Smail Niar 36

Testing the Resilience of Fail-Operational Systems Early On with Non-Intrusive Data Seeding
Joachim Fröhlich, Jelena Frtunikj, Alois Knoll 41

A HW-dependent Software Model for Cross-Layer Fault Analysis in Embedded Systems
Christian Bartsch, Carlos Villarraga, Dominik Stoffel, Wolfgang Kunz 43

Component Fault Localization using Built-In Current Sensors for Error Resilient Computation
Seetal Potluri, A. Satya Trinadh, Siddhant Saraf, Kamakoti Veezhinathan 50

Efficient Fault Emulation through Splitting Combinational and Sequential Fault Propagation
Ralph Nyberg, Johann Heyszl, Georg Sigl 52

Graph Guided Error Effect Simulation
Jo Laufenberg, Sebastian Reiter, Alexander Viehl, Oliver Bringmann, Wolfgang Rosenstiel 54

Towards Generating Test Suites with High Functional Coverage for Error Effect Simulation
Aljoscha Windhorst, Hoang M. Le, Daniel Große, Rolf Drechsler 58

Accurate Cache Vulnerability Modeling in Presence of Protection Techniques
Yohan Ko, Reiley Jeyapaul, Youngbin Kim, Kyoungwoo Lee, Aviral Shrivastava 60

On the Correlation of Hardware Faults and Software Errors
Wolfgang Mueller, Liang Wu, Christoph Scheytt, Markus Becker, Sven Schönberg 62

Aging Aware Timing Analysis Incorporated Into a Commercial STA Tool
Shushanik Karapetyan, Ulf Schlichtmann 66

An FPGA-based Testing Platform for the Validation of Automotive Powertrain ECU
Luca Sterpone, Davide Sabena, Lorenzo Venditti 68

Incremental System Design with Cross-Layer Dependency Analysis
Mischa Moestl, Rolf Ernst 75

ErrorPro: Software Tool for Stochastic Error Propagation Analysis
Andrey Morozov, Regina Tuk, Klaus Janschek 77

Fault Injection in Multi-Domain Physical System Models at Different Levels of Abstraction
Raghavendra Koppak, Oliver Bringmann, Andreas von Schwerin 79

Comparison of Different Fault-Injection Methods into TLM Models
Bogdan-Andrei Tabacaru, Moomen Chaari,Wolfgang Ecker,Thomas Kruse, Cristiano Novello 83

CLEAR: Cross-Layer Exploration for Architecting Resilience
Combining Hardware and Software Techniques to Tolerate Soft Errors in Processor Cores

Eric Cheng[1], Lukasz G. Szafaryn[2], Shahrzad Mirkhani[1], Hyungmin Cho[1], Chen-Yong Cher[3], Kevin Skadron[2], Mircea Stan[2], Klas Lilja[4], Jacob A. Abraham[5], Pradip Bose[3], Subhasish Mitra[1]

[1]Stanford University [2]University of Virgnia [3]IBM [4]Robust Chip, Inc. [5]University of Texas at Austin

***Abstract*—Many resilience techniques have been published in research literature and used during processor design to improve system reliability. These techniques span various layers of the system stack: circuit, logic, architecture, software, and algorithm layers. Architecting and embedding resilience often imposes large overheads in terms of energy/power, execution time, and area. In this paper, we use cross-layer resilience to effectively combine resilience techniques across the layers of the system stack to achieve a 50× Silent Data Corruption (SDC) improvement for a general-purpose computing system at 1.5× less energy as compared to an optimized single-layer (e.g., single-technique) approach. By injecting over 9 million flip-flop errors, we carefully characterize resilience techniques and validate our resilient designs to demonstrate our ability to finely tune resilience for both a simple, in-order SPARC processor and a complex, out-of-order Alpha processor.**

Keywords—cross-layer resilience, soft errors, reliability

I. Introduction

We address a significant challenge in the design of robust processor cores that are resilient to errors, specifically radiation-induced soft errors, in the underlying hardware: given a set of resilience techniques, what is the best way to protect a given processor design from soft errors using a **combination** of techniques, **across multiple abstraction layers** (e.g., cross-layer resilience [Carter 10, DeHon 10, Mitra 10]), such that the overall resilience targets are met at minimal costs (energy, power, execution time, area)?

We present a new framework called CLEAR (Cross-Layer Exploration for Architecting Resilience) for exploring a wide variety of soft error resilience techniques and their combinations in a systematic manner. Our cross-layer analysis differs from prior work [Szafaryn 13, Timor 10] that rely on abstracted views of resilience techniques (no detailed analysis) and thus cannot evaluate combinations systematically or accurately. Our framework contrasts current practice (e.g., commercial solutions from IBM [Meaney 05] and solutions like Argus [Meixner 07]), where cross-layer resilience is generally based on "designer experience" or "historical practice" and which lack fine-grained analysis.

We make the following contributions:

1. A framework for exploring the large space of cross-layer resilience of processors against soft errors and quantify the strengths and weaknesses of various combinations.
2. Demonstrate the generality of our results using two different processor designs.
3. Demonstrate that not all cross-layer resilience techniques are effective for soft error protection of processors (e.g., combining software, logic, and circuit layers could result in 15× more energy overhead than a circuit-layer only approach).
4. Our extensive study reveals two cross-layer approaches as highly effective solutions: (a) Combining circuit-level radiation hardening and logic-level parity checking, together with error recovery (b) Combining algorithm-level error correction together with circuit-level radiation hardening, logic-level parity checking, and micro-architectural cross-layer error recovery (which achieves a 1.5× improvement in energy overhead compared to an optimized circuit-layer approach).
5. Quantify benchmark sensitivities and a method for minimizing their impact.

II. CLEAR Framework

Our CLEAR framework (Fig. 1) is driven by: 1. Systematic characterization of various resilience techniques (Fig. 2) (e.g., circuit-level techniques are calibrated using radiation results from test chips); 2. Comprehensive (over 9 million) error injections using accurate error models on BEE3 FPGA-based emulation and University of Texas Stampede supercomputer-driven RTL simulations [Cho 13]; 3. Post-layout estimates of costs associated with various resilience techniques using industrial TSMC 28nm libraries (logic cells, SRAM) and Synopsys design tools; and 4. Thorough execution time and energy estimates for software-implemented resilience using FPGA-based emulation and detailed RTL simulations.

Two diverse processor designs (a simple in-order SPARC processor and a complex out-of-order and super-scalar Alpha processor were chosen as representative of *in-order cores (InO-cores)* and an *out-of-order cores (OoO-cores)*. These designs span application domains from embedded to high performance computing (HPC). We evaluate using the SPECINT 2000 and the DARPA PERFECT benchmark suites. Two metrics are used to evaluate resilience improvement: *SDC (Silent Data Corruption) improvement* and *DUE (Detected but Uncorrected Error) improvement* [Sanda 08].

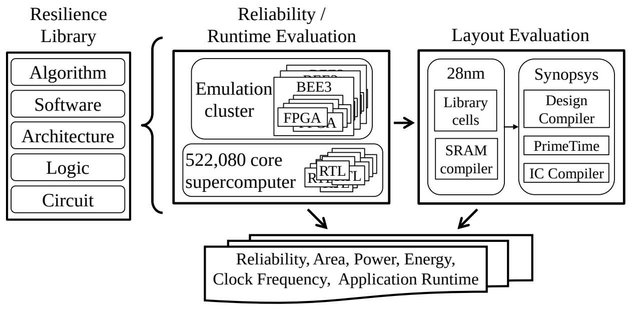


Figure 1. CLEAR Framework

Table 1. Resilience Technique Summary: Key Characteristics for an InO-core

Combination	50× SDC Improvement			50× DUE Improvement			Perf.
	Area	Power	Energy	Area	Power	Energy	
Hardening	2.9%	7.3%	7.3%	3.8%	9.4%	9.4%	0%
Harden + parity + cross-layer recovery	2.5%	6.1%	6.1%	3.6%	8.4%	8.4%	0%
EDS + harden + parity + cross-layer recovery	2.7%	6.6%	6.6%	3.8%	8.5%	8.5%	0%
DFC + harden + parity + DFC recovery	40.8%	37.2%	58.5%	42.2%	39.3%	59.0%	6.2%
ABFT correction + harden + parity + recovery	1.0%	1.7%	1.7%	1.5%	3.3%	4.3%	1.4%
ABFT correction + LEAP-ctrl + harden + parity + recovery	4.0%	2.6%	4.0%	4.1%	4.6%	6.2%	1.4%
DFC + harden + parity (no DFC recovery)	5.5%	6.2%	15.7%	6.2%	8.3%	16.4%	6.2%
Assertions + harden + parity + checkpoint-restart	1.2%	3.0%	19.1%	3.8%	9.0%	26.0%	15.6%
CFCSS + harden + parity + checkpoint-restart	1.4%	2.9%	44.5%	4.2%	9.5%	54.1%	40.6%
EDDI + harden + parity + checkpoint-restart	0.4%	0.6%	111%	4.0%	9.5%	130.1%	110%
ABFT detection + harden + parity + chkpt-rstrt	1.6%	3.4%	28.3%	1.6%	3.8%	28.7%	14.0%
DFC + ABFT correction + harden + parity + ckpt-rstrt	4.2%	3.6%	13.5%	3.8%	2.5%	12.1%	10.2%

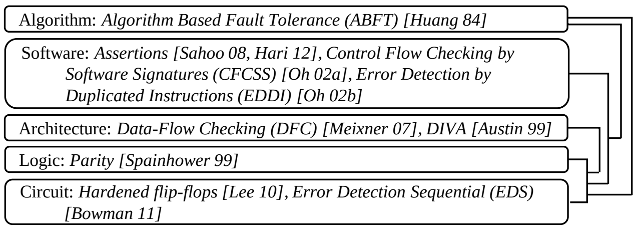

Algorithm: *Algorithm Based Fault Tolerance (ABFT) [Huang 84]*
Software: *Assertions [Sahoo 08, Hari 12], Control Flow Checking by Software Signatures (CFCSS) [Oh 02a], Error Detection by Duplicated Instructions (EDDI) [Oh 02b]*
Architecture: *Data-Flow Checking (DFC) [Meixner 07], DIVA [Austin 99]*
Logic: *Parity [Spainhower 99]*
Circuit: *Hardened flip-flops [Lee 10], Error Detection Sequential (EDS) [Bowman 11]*

Figure 2. Resilience Techniques and their Combinations

III. Cross-layer Case Study

Table 1 summarizes the various combinations of resilience we have explored and their costs for achieving a 50× improvement in SDC and DUE for an InO-core. We highlight the three best solutions: (hardening-only), (hardening, parity, recovery), and (ABFT correction, hardening, parity, recovery – both variants with and without the addition of LEAP-ctrl). Our results are consistent for the OoO-core as well.

IV. Benchmark Dependence

To evaluate the dependence of our results on particular benchmarks, we incorporate resilience using a set of benchmarks (a *training set*), and then evaluate resilience with respect to another disjoint set of benchmarks (an *evaluation set*). We found that, typically, resilience improvement is underestimated when considering different benchmarks.

We propose to minimize benchmark effects by applying our cross-layer resilience methodology as normal (based on the training set) and subsequently replacing all flip-flops left unprotected with a very inexpensive lightly-hardened flip-flop. We found that that using lightly-hardened flip-flops allows us to meet or exceed the desired resilience target at low (less than 1% additional chip-level) cost.

V. Conclusions

We have presented a new methodology and framework for exploring, evaluating, and optimizing cross-layer resilience. We have shown that using our methodology, one can find effective cross-layer combinations that can reduce the energy overhead required for resilience by 1.5× as compared to the best possible single-layer solutions. The most important insight is that coverage can be tuned, and the most efficient protection achieved, by optimizing at the level of individual flip-flops. The resulting coverage obviates the need for architectural and most software resilience techniques. Finally, we have demonstrated the important role benchmarking plays when evaluating the resilience of a system and proposed a low-cost solution for minimizing the effects of benchmark dependence.

Acknowledgment

This work was supported by DARPA, DTRA, SRC, and NSF.

References

[Austin 99] Austin, T.M., "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," *Intl. Symp. on Microarchitecture*," 1999.

[Bowman 11] Bowman, K., *et. al.,*, "A 45 nm Resilient Microprocessor Core for Dynamic Variation Tolerance," *Journal of Solid-State Circuits*, vol. 44, no. 1, pp.49-63, 2009.

[Carter 10] Carter, N., H. Naeimi, and D. Gardner, "Design Techniques for Cross-layer Resilience," *Design & Test Europe*, pp. 1023-1028, 2010.

[Cho 13] Cho, H., *et al.*, "Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design," *Design Automation Conference*, pp. 1-10, 2013

[DeHon 10] DeHon, A., H. Quinn, and N. Carter, "Vision for Cross-layer Optimization to Address the Dual Challenges of Energy and Reliability," *Design & Test in Europe*, pp. 1017-1022, 2010.

[Hari 12] Hari, S. K. S., *et. al.*, "Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults." *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, 2012.

[Huang 84] Huang, K, and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Computers*, C-33(6):518-528, 1984.

[Lee 10] Lee, H. K., *et. al.*, "LEAP: Layout Design through Error-Aware Transistor Positioning for Soft-Error Resilient Sequential Cell Design," *Intl. Reliability Physics Symposium*, pp. 203-212, 2010.

[Meaney 05] Meaney, P. J., *et. al.*, "IBM z990 Soft Error Detection and Recovery," *Trans. on Device and Materials Reliability*, pp. 419-427, 2005.

[Meixner 07] Meixner, A., M. E. Bauer, and D. J. Sorin, "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," *International Symposium on Microarchitecture*, vol., no., pp.210-222, 2007.

[Mitra 10] Mitra, S., K. Brelsford and P. Sanda, "Cross-Layer Resilience Challenges: Metrics and Optimization," D*esign Automation and Test in Europe*, 2010.

[Oh 02a] Oh, N., P.P. Shirvani, and E.J. McCluskey, "Control Flow Checking by Software Signatures," *Trans. Reliability*, vol.51, no.1, pp.111-122, 2002.

[Oh 02b] Oh, N., P.P. Shirvani, and E.J. McCluskey, "Error Detection by Duplicated Instructions in Super-Scalar Processors," *Trans. Reliability*, vol. 51, no. 1, pp. 63-75, 2002.

[Sahoo 08] Sahoo S. K., *et. al.*, "Using likely program invariants to detect hardware errors." *IEEE Dependable Systems and Networks*, 2008.

[Sanda 08] Sanda, P. N., *et. al.*, "Soft-error resilience of the IBM POWER6 processor," *IBM Journal of Research and Development*, vol. 52, no. 3, pp. 275–284, May 2008.

[Spainhower 99] Spainhower, L. and T.A. Gregg, "IBM S/390 Parallel Enterprise Server G5 fault tolerance: A historical perspective," *IBM Journal of Research and Development*, vol. 43, no. 5.6, pp. 863-873, 1999.

[Szafaryn 13] Szafaryn L. G., B. H. Meyer, and K. Skadron, "Evaluating Overheads of Multibit Soft-Error Protection in the Processor Core," *IEEE Micro*, vol.33, no.4, pp. 56-65, July-Aug. 2013.

[Timor 10] Timor, A., et al., "Using Underutilized CPU Resources to Enhance Its Reliability," *Dependable and Secure Computing, IEEE Transactions on*, vol.7, no.1, pp.94,109, Jan.-March 2010.

Cross-layer Resilience Mechanisms to Protect the Communication Path in Embedded Systems

Tobias Stumpf, Hermann Härtig
Operating Systems Group
TU Dresden, Germany
{tstumpf|haertig}@tudos.org

Eberle A. Rambo, Rolf Ernst
Institute of Computer and Network Engineering
TU Braunschweig, Germany
{rambo|ernst}@ida.ing.tu-bs.de

***Abstract*—With the decreasing feature size of new chip generations, additional protection mechanism are necessary especially to protect safety-critical systems in environments with higher radiation levels. This paper investigates a cross-layer approach combining hardware and software-level techniques into a complementary protection mechanism. This reduces overhead by avoiding overlapping protection without reducing the fault tolerance. The paper starts by introducing one software and one hardware protection mechanism. Then, we discuss the overlap as well as pros and cons of both techniques. Finally, we give an outlook about possible benefits to combine both independent approaches to one cross-layer resilience mechanism.**

I. Introduction

Technology scaling influences the system design and reliability [1]. Because increasing the single thread performance reached its limits, hardware vendors increase the parallelism by scaling up the core count in order to keep increasing performance. This has been enabled by decreasing the feature size, providing several benefits like reduced power consumption and increased transistor density on the one hand. On the other hand, it increases the susceptibility to soft-errors [2], which extends to the whole chip, including the interconnection between the cores, giving rise to the so-called unreliable hardware.

The unreliability can be addressed in software as well as in hardware. In hardware, the effects of such soft-errors are abstracted as bit-flips in registers and memory. In software, it can be abstracted as data corruption or misbehaving execution. Software-based approaches, like replication, checkpoint restart, or encoded processing, can overcome soft-errors or bring the system into a fail-safe state.

A remaining weak point is the inter-core communication. The software layer can add redundancy to detect data corruption, but it is unaware of undelivered messages. Moreover, parts of the message are not visible to software and must be handled at the hardware level. Detecting an error at software level also leads to additional overhead, compared e.g. to error detection and retransmission at the hardware level.

A particle strike can cause single or multiple bit-flips [3]. With decreasing feature size the probability for multi bit-flips, caused by one particle strike, increases. Because ECC can recover only from single bit flips, the messages can be additionally protected in software, which guarantees the message integrity but not its delivery.

In this paper, we investigate how soft-errors, in the form of Single Event Upsets as well as Multi Event Upsets, can impair the whole communication path of a multi-core platform and, more importantly, how to protect it. A cross-layer resilience approach to the problem is discussed, where the protection in software and hardware can maximize the fault-tolerance while avoiding unnecessary accumulated overhead.

II. Related Work

Fault tolerance can be achieved at different abstraction layers. The physical layer ensures the correct transmission between two transmission points, whereas the network or transport layer includes further techniques to ensure that a message is finally delivered to the end point. Because of the wide-spread use of TCP/IP, several fault tolerance approaches exist to extend its reliability. The existing mechanisms focus on different layers. Song et al. [4] developed a fault-tolerant Ethernet protocol using COTS hardware, which is application level transparent. Their approach extends the network stack and adds the fault tolerance functionality between network data link layer and the transport and network protocol layers. Other techniques, like PortLand [5], extend the data link layer. Because higher level protocols like TCP/IP also include fault-tolerance mechanisms, the fault-tolerance overhead is probably higher than necessary.

An overall system approach is made by the Tandem Non Stop technology [6]. It is based on special purpose hardware and the system software was developed with fault-tolerance in mind. Because of the high development effort, the successors of the original Non Stop system goes into the direction of COTS hardware.

Combining hardware and software mechanisms can reduce the fault-tolerance overhead or can improve the overall system performance. Kariniemi and Nurmi [7] designed a NoC where the hardware and software protocols are tightly coupled to increase communication performance. They also considered fault-tolerance in their work.

In the following, we will focus on fault-tolerance to increase the robustness, by decreasing the necessary costs.

III. System Overview & Fault Model

Figure 1 illustrates the system we will harden and which is used for evaluation. On the lowest layer, there are several CPUs which are interconnected. We assume that we have some CPUs which are more reliable and executing critical software parts, according to the design presented by Engel and Döbel [8]. If we cannot use hardened CPUs, then additional techniques like AN-Encoding [9] are necessary to detect a malfunctioning CPU.

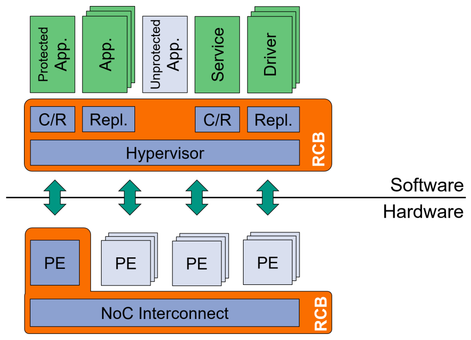


Fig. 1: Many-core system overview.

At software level, we use a microkernel based system. The highlighted part of Figure 1, the RCB, is the critical part which has to be executed on one of the reliable cores. The other software components are protected by existing techniques like Checkpointing/Restart [10] or replication [11].

In this work, we are focusing on transient faults, caused by events such as particle strikes or electromagnetic radiation. Due to its transient nature, after re-executing or rewriting the fault disappears. Moreover, an existing fault, visible at hardware level, may not influence the software at all because it is masked by the hardware or the faulty function unit or memory area is not used at all.

In the sequence, we analyze the failures caused by transient faults at the system level. Next, we discuss the impacts of faults on the different IPC communication steps. In Section VI, we discuss handling the faults in software. In Section VII we extend the analysis to the hardware level and discuss handling faults in hardware.

IV. System Analysis

To examine the failure-vulnerability of our system we performed a full fault analysis for core system functions of the used microkernel. Figure 2 illustrates the results. We grouped the results in four classes: Fault free (OK), system crashes and stops working (CRASH), the system continues work, but the outcome differs, which is called silent data corruption (SDC), and the system does not reach a specific execution point within a specified time frame (TIMEOUT).

System crashes and timeouts can have various reasons. A bit-flip my affect the system state and changes the execution so that an exception is triggered or the execution hangs in an endless loop. Those failures need different memory protection mechanism, which are out of the scope of this paper.

A crash could be the result of a wrongly delivered message and a timeout the result of a non-delivered message. Both cases will be discussed in Section VII.

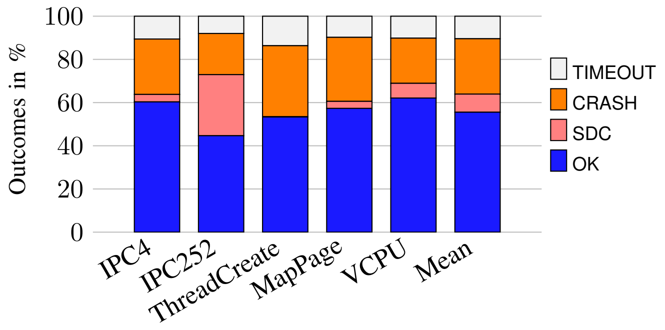


Fig. 2: Fault-injection experiments with L4/Fiasco.OC. Each bar illustrates one experimental setup and covers a basic microkernel functionality.

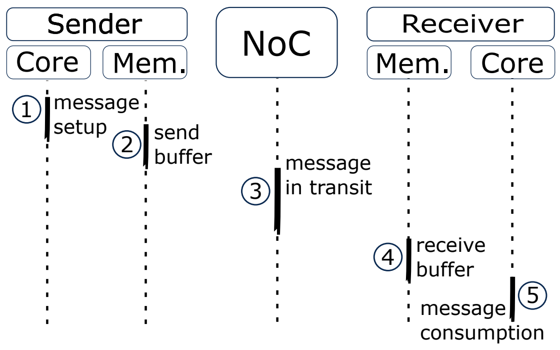


Fig. 3: Steps for delivering a message from client to server.

Further, a crash may also be the result of corrupted message, which leads to a wrong system execution. A detailed analysis of the SDC outcome of the performed test cases indicates that all non-detected failures leading to a wrong system output are the result of corrupted messages.

The next section describes how a message is delivered and explains the different fault cases.

V. IPC Communication and Fault Cases

The inter-process communication in a multi-core system is illustrated in Figure 3. In (1) , a client creates a message and hands over the message to the OS (hypervisor). During these steps, the message is stored in memory (2). In (3), the message is then transferred through the interconnect (the Network-on-Chip) to the receiver, where it is stored in a buffer (4). The OS then hands the message over to the other application (5). Depending on the implementation, the message may be transferred to main memory before being read by the receiver, in which case, the message would go through the NoC and memory once more.

Soft-errors can occur in any of the steps of an inter-process communication and therefore result in different failures. The communication can be protected in hardware (NoC), which guarantees the delivery and integrity of messages. A problem arises when an error corrupts the message before being handed over to the NoC (2), or similarly, after being delivered but before being consumed (4). Even though it is a common practice to protect the memory with ECC, it is not sufficient [12], especially for future systems with decreased feature size.

VI. Software Mechanism

At software level only some parts of the faults are visible, because some errors are masked by the hardware. Faults manifesting into an error can lead to different results: Malfunctioning software or data corruption. The execution path may be changed because the input values differ, which can also result in a system crash. Other bit-flips in the network packet may influence the outcome of a calculation, which cannot be detected.

For off-chip communication, checksums are state-of-the-art to protect message header and payload. But message transmission within the same die or even across package boundaries of the same system is not appropriately protected [13] or assumed to be fault-free. We expect that the reliability decreases with future systems and makes a message protection within the same die necessary. Therefore, we evaluated a software based approach.

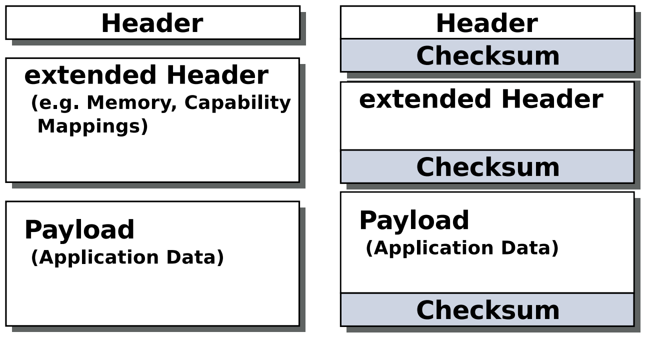
A. Systems Software & Messages

To harden the communication, we looked at a microkernel-based operating system. This is in line with the presented system design in Section I, because the microkernel is part of a small reliable computing base and most software parts can run on top of it, including system software services. In Section V we gave a general overview of the inter process communication. In this section, we focus on the software's point of view.

The simplest form is data exchange. One process creates a message and initiates the transfer. Then, the kernel transfers the data and informs the receiver about the incoming message. For data transfer, a message is composed of two parts, the header (including the receiver, information about the message type, size, etc.) and the payload with the data.

But IPC is not only used to transfer data between sender and receiver. For instance, it is also used to grant access rights. A process A can grant process B access rights to some part of its memory by sending a specific message M_S. The message M_S includes information about the memory area and the rights. Process B has to be ready to receive this information. Therefore, it creates a special message M_R, telling the kernel that it is ready to receive the memory mapping and specifies, at which point its able to receive the mapping.

In addition to communication, IPC messages can be used for OS specific operations, which goes further than simple data transmission. Figure 4a illustrates a simplified version of an IPC message. The extended header is not transferred from the sender to the receiver. Instead, the kernel interprets the information and creates memory mappings or grants stigates a cross-layer +approach combining hardware and software-level techniques into a complementary +protection mechanism. This reduces overhead by avoiding overlapping protection +without reducing the fault tolerance. The paper starts by introducing one +software and one hardware protection mechanism. Then,



(a) unprotected (b) protected

Fig. 4: Comparison of an unprotected and a protected IPC message.

we discuss the overlap as +well as pros and cons of both techniques. Finally, we give an outlook about

access rights.

First of all, we will focus on data transfer only and present our general approach and first measurements. We give an outlook on the advanced part at the end of this section.

B. Message Protection

A closer look at the results shown in Figure 2 indicates that all SDC corruption is caused by a corrupted message. Using one checksum for the whole IPC package is not feasible, because an IPC packages is modified during transmission, which requires several checks and recalculation during package delivery. Therefore, we added one dedicated checksum for the payload.

To reduce the amount of additional read operations, the checksum is calculated during message creation and finally appended to the message after the last part of the payload is written. We choose two different checksum algorithms: CRC32 and parity byte. In general, CRC32 can deal with more fault cases than a simple parity byte, but the CRC32 checksum is more expensive to calculate.

C. Evaluation

Figure 5 presents the runtime overhead for different implementations. For the measurement, we sent one message with the maximum payload from one thread to another, which gives us an upper bound for the overhead. Our first software implementations of the CRC32 slows down the transmission by a factor of 8 compared to the small overhead of less than 17% percent of a simple parity bit. Therefore, we analyzed two alternatives: A fast software implementation using a table with pre-calculated values. Besides the memory overhead, this solution has the drawback, that the pre-calculated table is again vulnerable to bit-flips. The second alternative uses hardware extensions.

We repeated the experiment with the highest SDC from Figure 2 for the different implementations. For the repeated experiment we see no SDC at all, because the IPC252 experiment sends only data which is now fully protected. The drawback

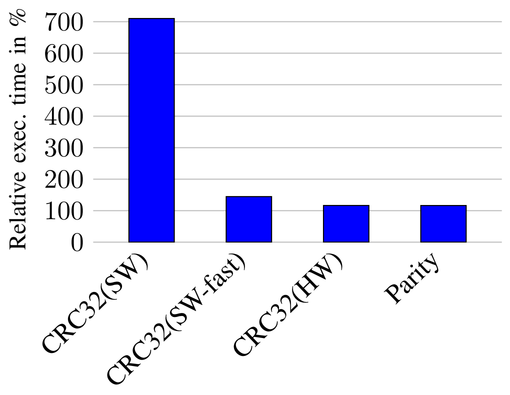


Fig. 5: Runtime overhead for different IPC protection mechanisms.

of our experiments is an increased amount of timeouts and crashes.

D. Outlook

To address timeouts and crashes caused by corrupted messages, we currently harden the remaining part of an IPC message. Because the different parts of an IPC message are read or modified at different points, we group the data which is used together. One possible solution is illustrated in 4b, but we will investigate if a more fine grained approach can minimize the protection overhead. If the hardware provides instructions to calculate the CRC32, we will use the hardware-based approach, because it provides the best ratio between overhead and error-detection. For older or cheaper processors without support for CRC32 calculation, we will use a simple parity byte.

The presented software approach addresses failures, which are visible as memory corruption at software level. But it requires that all messages are finally transmitted and the receiver gets notice about the incoming message. If packets, including interrupts, can be lost, further techniques are necessary. In software, we can implement a more complex communication protocol with timeouts and retransmissions. However, we can also design a reliable on-Chip network, to reduce the fault-tolerance overhead caused by additional software. We will discuss such an approach in the next section, followed by a discussion about combining software and hardware techniques.

VII. Hardware Mechanism

The error detection of a software-based approach is limited to delivered packets, including notifying the receiver. If a message is lost while being forwarded by the hardware or delivered to the wrong core, an application may stall forever. Using timeouts in software, also known as watchdogs, solves the problem in some cases while incurring additional delays in the worst-case. In other cases, it does not help. For instance, when forwarding interrupts, which are a special type of message and are related to signals/exception at the software

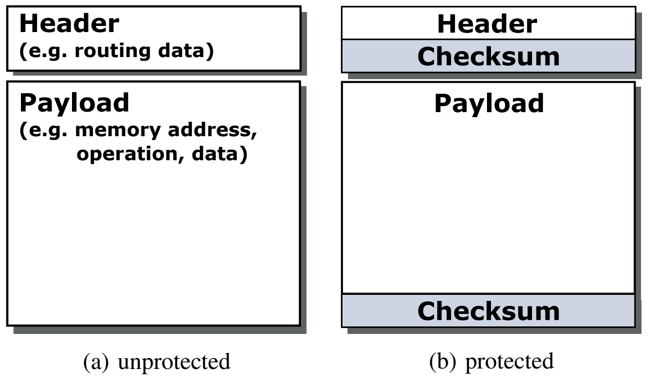


Fig. 6: Comparison of unprotected and protected NoC packets.

layer. This type of communication is not explicit in software and thus must be handled in hardware.

The NoC protection against transient faults must consider two aspects: the control and the data. The control concerns the network availability, i.e. its capability to restore service after an error affects a router's state machine. The data concerns packet delivery and packet payload integrity. Here, we focus on the latter. We focus on errors affecting data. Therefore, we assume that errors affecting the availability of the network are either handled by an orthogonal approach, such as resilient state machines at design time or error detection and recovery at the component level. We assume that errors affecting control may lead to packet loss, e.g. due to a component reset or irrecoverable routing data. On an end-to-end perspective, transient faults may cause packet loss (i.e. affecting packet delivery) or cause packet corruption (i.e. affecting the payload integrity). These are addressed next.

A. Packet integrity

The impacts of packet corruption depend on whether the packet header or its payload is affected. The unprotected packet is illustrated in Figure 6a.

The packet header contains the routing data, responsible for delivering the packet to the right destination. The payload contains everything else that is not related to routing, e.g. memory address for a memory transaction, access rights, and the data being transferred. The entire IPC message (Figure 4) is here part of the payload of a packet in the NoC. The rest of the payload contains memory address and operation type. Depending on the IPC size, it may be divided and transported as the payload of several packets. The handling of IPCs in hardware and software will be detailed in Section VIII.

We protect the header of the packet and its payload separately. This is shown in Figure 6b. The header's integrity is checked in each router before processing the packet in that router. The payload's integrity is checked in the receiver's network interface. This allows the header to be quickly checked and updated in the routers without requiring the whole packet to be received and processed, as seen in faster wormhole-switched networks [14]. In these networks, the packets are composed of Flow Control Units (flits) and each flit has a

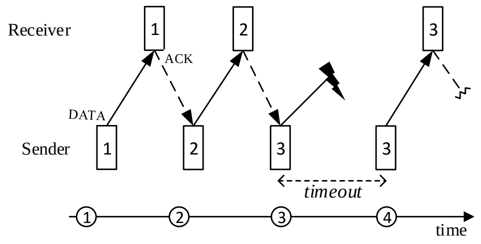


Fig. 7: Illustrative example of Stop-and-Wait ARQ.

header which is encoded such that each flit has a checksum and is checked separately; the payload is distributed among the flits has only one checksum that covers the entire payload (not shown in the Figure).

B. Packet delivery

To provide reliable transmission of data in the NoC without incurring high area overhead in hardware, we implement end-to-end transport protocols. These protocols may be selected and configured by software through registers in the network interface. In this work, we consider ARQ retransmission protocols, such as Stop-and-Wait and Go-Back-N [14]. Other transport protocols, such as Multipath Routing (MPR) [15] and other optimized ARQ protocols, can also be used. Disabling the reliable packet delivery is also possible, for instance in the case of sensor readings, whose readings may be occasionally lost (packet integrity however must be ensured).

ARQ-based protocols are widely used to guarantee packet delivery [14]. Its simplest variant is Stop-and-Wait, illustrated in Figure 7. For each packet sent $\textcircled{1}$, the sender node waits for an acknowledgement from the receiver node before sending the next packet $\textcircled{2}$ or retransmitting after a timeout $\textcircled{4}$, in case of error $\textcircled{3}$. The throughput is improved in Go-Back-N, which allows n packets to be sent (the *send window*) before stopping and waiting for an acknowledgement [14].

C. Evaluation

We evaluated the mechanism with fault injection experiments. The NoC was modeled and simulated in OMNeT++. The traffic was generated by applications from the benchmark MiBench [16] instantiated in a 3x3 2D-mesh NoC [17]: susan and qsort in Double Modular Redundancy (DMR), blowfish, bitcount. Errors were injected randomly in the entire NoC with a varying Bit Error Rate (BER).

First, we evaluate the impact of errors on the NoC without a hardware mechanism to guarantee the packet delivery. Figure 8 shows the packet delivery (considering integer packets) as the BER increases. Because of corruption and errors in routers, more packets are dropped as the BER increases. The BERs utilized in the experiments are artificially high to explore the performance of the approach under stress and lower the simulation time. In practice, soft errors do not occur so often (BER$< 10^{-9}$).

Now, we evaluate the performance of the transport protocol to guarantee packet delivery. Stop-and-Wait ARQ was employed as the transport protocol. Figure 9 shows the variation

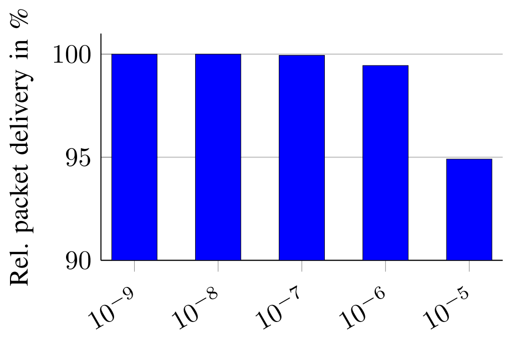


Fig. 8: Impact of errors on MiBench traffic as BER increases. Variation of packet delivery w.r.t. the error free scenario.

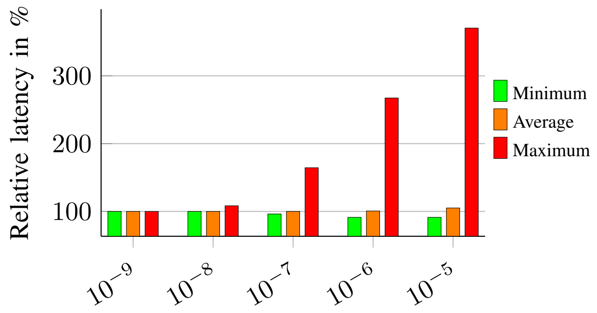


Fig. 9: MiBench traffic latency as BER increases on an Stop-and-Wait ARQ transport protocol. Variation of latency w.r.t. the error free scenario.

in latency when increasing the BER, relative to the error free scenario. For each BER, the plot shows the mean value over the variations of latency from all application instances w.r.t. the error free scenario. ARQ is able to deliver all packets as the error rate increases (guaranteed delivery). Since it employs retransmission to do so (which includes timeouts), the impact of increasing error rates is seen in the maximum latencies. Despite that, the average latency almost does not increase, since errors are not the general case but exceptions. Interestingly, the minimum latencies decrease. This is due to the fact that packets are dropped because an error (cf. Figure 8) causing a temporary decrease of traffic interference for some packets.

D. Outlook

The presented hardware approach is able to handle soft errors occurring in the NoC and affecting any communication. The protection, although transparent to the software execution, is configurable by software, which can select the transport protocol employed. It covers all explicit communication, such as IPCs and shared memory accesses, and also implicit communication, which are usually transparent to software, such as cache line misses, memory coherence protocol messages, and interrupt forwarding.

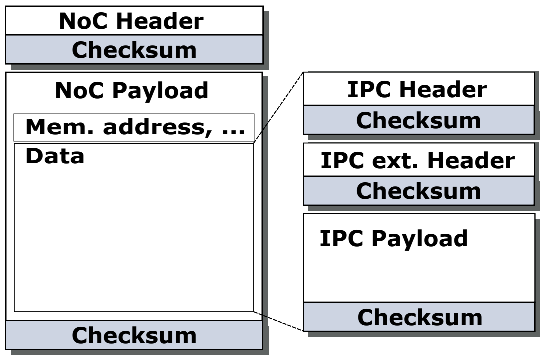


Fig. 10: The relation between the protected NoC packet and the protected IPC message.

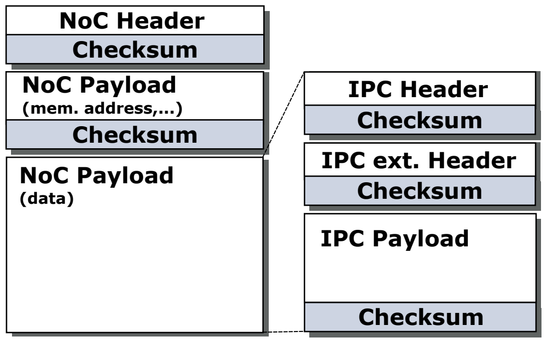


Fig. 11: Optimized protection of a NoC packet for IPC traffic. Checksum is calculated in software.

VIII. Hardware vs. Software

After reading Sections VI and VII, one may notice that the data protecting is overlapping. Both hardware and software protect the transmitted data with checksums. This is illustrated in Figure 10. The IPC message is protected by software and also protected by an additional checksum in hardware. Now a question can be raised: should the protection for IPCs be removed from hardware or from software?

Can we remove the software protection? After delivery, the message remains in the local memory and is still prone to errors until the data is consumed.

Can the hardware protection be removed? Some parts of the IPC message is not visible to software, e.g. memory address and other control information, and has to be protected in hardware. This cannot be neglected as it can lead to memory corruption inside the RCB, a case of error propagation. Other parts can be protected in software, e.g. the IPC data itself. Moreover, IPC messages are only part of the NoC traffic. Non-IPC traffic (e.g. interrupt delivery) has to be protected in hardware.

Finally, it is not a question of either-or. It is about how both techniques can cooperate in synergy for the sake of performance and efficiency. We will investigate the possibility of disabling the hardware protection when transmitting messages already protected by software.

A possible solution is to disable the hardware checksum for data when transmitting IPC messages. The integrity check is then only performed in software avoiding double overhead. The solution is illustrated in Figure 11, where the data part of the payload is not covered by the checksum in hardware. The special packet format is configured in the network interface of the sender and applies only to IPC messages (differentiated through the memory address).

Let us reason about the benefits. Considering that a packet in the NoC is able to transport 32 Bytes of data and assuming that it would be protected by a checksum 10 bits long (e.g. CRC-10 [18]), the approach would avoid CRC generation and check for these packets and reduce the amount of transmitted data in 4%. Increasing the packet length to transport 64 Bytes of data and increasing the checksum to 12 bits (e.g. CRC-12 [18]),

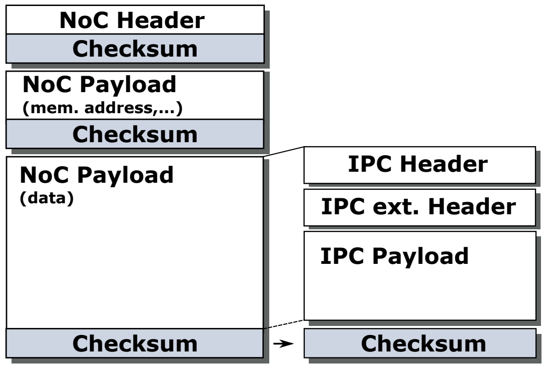


Fig. 12: Optimized protection of a NoC packet for IPC traffic. Checksum is calculated in hardware and checked in software.

the reduction of the amount of transmitted data decreases to 2%. Since data corruption will not be detected in the NoC, the retransmission or re-execution of the IPC in case of errors is handled in software.

Another possible approach is to delegate the task of calculating the IPC checksums to hardware. The sender node creates the IPC message in a local memory. After creation, the message is immediately sent through the NoC. The NoC is then responsible for creating the checksum for the whole IPC message, which takes place together with the transmission. At the destination, the checksum used in the NoC is written to the memory at a specified position in the end of the IPC. The solution is illustrated in Figure 12. The NoC checksum for the IPC is also delivered to the IPC message receiver, which uses it to check the message integrity before consuming it. The check is performed in software.

With this approach, the software overhead can be reduced from 17% to the overhead caused by the hardware implementation, which is one order of magnitude lower. But this benefit comes with the drawback of more complex hardware design, including the hardware implementation of e.g. CRC-32 in each network interface, which increases the chip size and therefore production costs as well as energy consumption.

IX. Conclusion

We have investigated the communication path in an embedded system and how it can be protected to be resilient to soft errors. A software approach can protect data until usage without special hardware. However, it has the drawback of higher overhead in time when compared to a hardware one. Hardware-based protection on the other side increases the production costs because special hardware is needed. A cross-layer approach combining hardware and software techniques can increase the fault-tolerance without prohibitive overheads in time and area. We have discussed initial ideas towards an efficient cross-layer solution, opening the path for future research.

X. Acknowledgement

This work was partly founded by German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 – spp1500.itec.kit.edu) and the cluster of excellence Centre for Advancing Electronics Dresden (cfAED).

References

[1] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, no. 5, pp. 67–77, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1941487.1941507

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

[3] R. A. Reed, M. A. Carts, P. W. Marshall, C. J. Marshall, P. J. Musseau, O.and McNulty, D. R. Roth, S. Buchner, J. Melinger, and T. Corbiere, "Heavy ion and proton-induced single event multiple upset," *IEEE Transactions on Nuclear Science*, vol. 44, pp. 2224–2229.

[4] S. Song, J. Huang, P. Kappler, R. Freimark, and T. Kozlik, "Fault-tolerant ethernet middleware for ip-based process control networks," in *Proceedings 27th Conference on Local Computer Networks, Tampa, Florida, USA, 8-10 November, 2000*, 2000, pp. 116–125.

[5] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: A scalable fault-tolerant layer 2 data center network fabric," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 39–50, Aug. 2009.

[6] J. F. Bartlett, "A nonstop kernel," in *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, ser. SOSP '81. New York, NY, USA: ACM, 1981, pp. 22–29. [Online]. Available: http://doi.acm.org/10.1145/800216.806587

[7] H. Kariniemi and J. Nurmi, "Noc interface for fault-tolerant message-passing communication on multiprocessor soc platform," *NORCHIP*, 2009.

[8] M. Engel and B. Döbel, "The reliable computing base - a paradigm for software-based reliability." in *GI-Jahrestagung*, 2012, pp. 480–493.

[9] U. Schiffel, M. Süßkraut, and C. Fetzer, "AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware," in *SAFECOMP '09: Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 283–296.

[10] J. S. Plank, "An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance," University of Tennessee, Tech. Rep. CS-97-372, July 1997.

[11] B. Döbel, "Operating System Support for Redundant Multithreading," Ph.D. dissertation, TU Dresden, 2014.

[12] A. Hwang, I. Stefanovici, and B. Schroeder, "Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 111–122. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150989

[13] E. Rambo, A. Tschiene, J. Diemer, L. Ahrendts, and R. Ernst, "Fmea-based analysis of a network-on-chip for mixed-critical systems," in *Networks-on-Chip (NoCS), 2014 Eighth IEEE/ACM International Symposium on*, Sept 2014, pp. 33–40.

[14] A. Tanenbaum and D. Wetherall, *Computer Networks*. Pearson Prentice Hall, 2011.

[15] S. Murali, D. Atienza, L. Benini, and G. De Michel, "A multi-path routing strategy with guaranteed in-order packet delivery and fault-tolerance for networks on chip," in *Proceedings of the 43rd annual Design Automation Conference*. ACM, 2006, pp. 845–848.

[16] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *WWC-4. 2001*, Dec 2001.

[17] B. Motruk, J. Diemer, R. Buchty, R. Ernst, and M. Berekovic, "IDAMC: A Many-Core Platform with Run-Time Monitoring for Mixed-Criticality," in *HASE*, 2012.

[18] P. Koopman and T. Chakravarty, "Cyclic redundancy code (crc) polynomial selection for embedded networks," in *Dependable Systems and Networks, 2004 International Conference on*. IEEE, 2004, pp. 145–154.

Reliability-Aware Task Mapping on Many-Cores with Performance Heterogeneity

Kuan-Hsun Chen[1], Jian-Jia Chen[1], Florian Kriebel[2], Semeen Rehman[2], Muhammad Shafique[2] and Jörg Henkel[2]
[1]Department of Informatics, TU Dortmund (TUD), Germany
[2]Department of Informatics, Karlsruhe Institute of Technology (KIT), Germany
Corresponding Author's Email: kuan-hsun.chen@tu-dortmund.de

I. Introduction

Due to the architectural design, process variations and aging, individual cores in many-cores systems exhibit heterogeneous performance. In terms of the architectural design, such as ARM big.LITTLE architecture [1], [3] integrates different types of cores with the same instruction sets but different frequencies in the system to accommodate the performance requirement with tolerable chip temperature or power consumption. When considering the countermeasure for soft errors on many-cores, a commonly adopted technique is Redundant Multithreading (RMT) [8] that achieves error detection and recovery through redundant thread execution on different cores for an application. However, with the performance heterogeneity, how to achieve the resource-efficient reliability becomes a non-trivial problem, since *Task mapping* and *Determining the task execution mode* (i.e. a task executes in a reliable mode with RMT or unreliable mode without RMT) both are susceptible to the resiliency of tasks and the performance of cores. A straight-forward solution could be a greedy mapping of reliability-critical task onto a high-frequency core like we adopted in *dTune* [6]. However, such a greedy approach would lack efficiency as it suffers from its local decisions because the reliability degradation for each task does not necessarily proportional to the cores' frequency degradation. In addition, it cannot provide any guarantee with respect to the satisfaction of deadline miss rate. We provide an example and demonstrate that it is not always reliability-wise beneficial to assign the reliability-critical task to the highest-frequency core.

As shown in Fig. 1, in this paper we explore how to efficiently allocate the tasks onto many-cores by using RMT to improve the overall dependability, with respect to both timing and functional correctness while also accounting for application tasks with *multiple compiled versions*. Such multiple reliable versions can be generated by using the reliability-aware compilers like [2] and [7], exhibiting diverse performance and reliability properties. By applying multiple reliable task versions and RMT, we are able to exploit the optimization space at both software and hardware-levels while exploring different area, execution time, and achieved reliability tradeoffs. The timing correctness can be defined as the deadline miss rate, which is typically adopted as the quality of service (QoS) metric in many practical real-time applications.

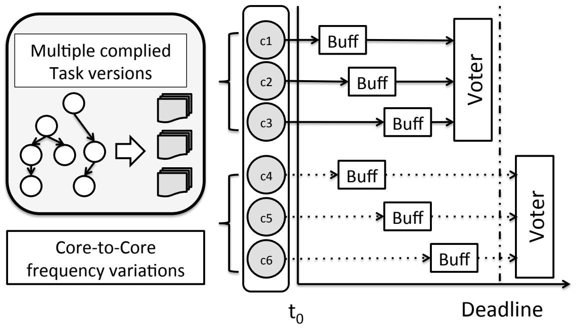

Multiple complied Task versions
Core-to-Core frequency variations
c1
c2
c3
c4
c5
c6
Buff
Buff
Buff
Buff
Buff
Buff
Voter
Voter
t0
Deadline

Fig. 1: Overview of interplay among performance heterogeneity, multiple task versions, and redundant multithreading. An example illustrates improper assigned cores group or version may lead to the deadline missing.

II. Problem Definition

Assume we are given a many-cores processor, which only has single thread per core, with ISA-compatible homogeneous RISC cores, and a set of tasks with multiple versions. The studied problem can be divided into two sub-problems, task mapping and execution mode adaptation. For the task mapping, assume we are given the execution modes and tolerable timing constraints, we consider *how to select the execution version and allocate the cores with different frequencies*, so that the overall reliability penalty is minimized. We observe that the problem can be connected to the well-known minimum weight perfect bipartite matching problem (MWPBM) and solved efficiently. The second sub-problem is the execution modes adaptation. The objective is to *determine the task execution modes without violating the satisfaction of deadline miss rate.* Without checking all the combinations, we propose an iterative mode adaptation to efficiently determine the execution modes of tasks with our mapping approaches so that the overall reliability penalty is minimized.

For the simplicity of presentation, the above approaches are all presented without data dependencies. After going through the approaches ideally, we discuss about how to incorporate the overhead of execution time for the data dependencies and communication with the model enhancement. Under the resource-constrained scenarios, the discussed scope of problem limits that the number of available cores is greater than or equal to the number of tasks without loss of generality.

III. Methods

For the systems which require only a homogeneous execution mode for all the tasks (i.e. either all tasks with RMT or without RMT), we reveal that this problem can be connected to MWPBM. According to the perfect matching property, we can adopt Hungarian Algorithm to deliver a feasible mapping for the tasks and cores, where each core only be assigned to one task. When the tasks have the heterogeneous execution modes (i.e., some tasks can execute with RMT and some cannot be supported in RMT due to the resource-constraint), we propose an efficient approach to assign the tasks onto the cores to achieve the higher system reliability.

After addressing the task-mapping problem, we consider how to decide the suitable execution modes under the resource-constraints. Again, the greedy strategy is not that beneficial. Some of tasks may suffer from their higher vulnerability, whereas some may suffer from their tighter tolerance of timing constraint. As it is not possible to check all the combinations of execution modes, we propose an iterative approach exploiting our task mapping approaches as the subroutine to guarantee the feasibility and efficiency of execution modes. We also consider how to model the communication between the individual tasks and in the redundant threads. By using software pipelining, we can quantify the maximum communication overhead among all the dependent tasks under the communication fabric with XY routing on 2-Dimension mesh, in which all the redundant cores are utilized concurrently.

IV. Results and Discussion

To evaluate the performance of our schemes fairly, we use the same setting in *dTune* [6] to obtain the reliability penalty of tasks by using a a real-world embedded benchmark MiBench and many-cores simulator for LEON3 ISA. The value of reliability penalties are obtained under fault rate 10^{-6} (in the unit of $\#fault/cycles$) to realize the high fault scenarios as adopted by the related works [4, 5]. We set the timing constraints as miss rate ρ. We normalize our results to the greedy mapping and compare the efficiency with the same set of tasks versions and core configurations, in which the normalized ratio is calculated as the resulting solution divided by the result of greedy mapping. By definition, the lower normalized penalty ratio is better.

The preliminary evaluation is performed by Grouping Frequency Levels with variations ω for evaluating architectures with heterogeneous performance, e.g., ARM big.LITTLE architecture [1]. We evaluated four different frequency levels in a multi-core processor. Assume the performance variation is ω, where the cores are with frequencies f_1, $(1-\omega)f_1$, $(1-2\omega)f_1$, and $(1-3\omega)f_1$, in which ω is up to 30% [3] due to the real-word scenarios on performance variations. As shown in Fig.2, we can observe that our approach outperforms the greedy mapping significantly. If the task mapping is not decided properly, the total reliability penalties will be increased dramatically. Since the difference of frequencies between different grouping levels is large enough, the greedy mapping may suffer from the sequential assignment of cores, in which the RMT tasks

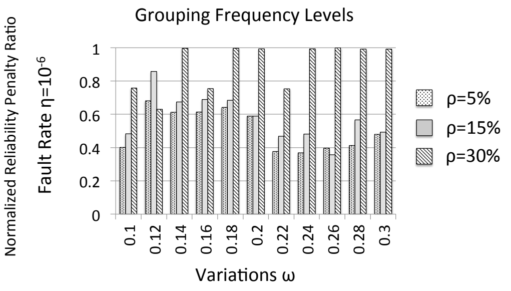


Fig. 2: Comparing the reliability penalty ratio by normalizing our result to the greedy mapping under 10^{-6} fault rates.

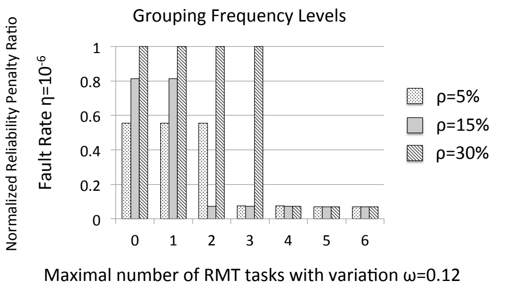


Fig. 3: Evaluation of the execution modes adaptation with variation $\omega = 0.14$ under 10^{-6} fault rates.

may have serve performance degradation. For the execution mode adaptation, we can see that the trends in Fig. 3 with the delivered execution modes still follow the previous observation in the task mapping. If the frequencies variation among the cores is not negligible as the case of grouping frequency levels, the effectiveness of proposed mapping approach is illustrious.

After all, we can conclude our proposed approaches provide a better overall reliability in terms of greedy strategy without violating both software and hardware-levels constraints.

Acknowledgments

This work is supported in parts by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 - spp1500.itec.kit.edu).

References

[1] ARM. big.little technology: The future of mobile, 2013.

[2] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri. A c/c++ source-to-source compiler for dependable applications. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 71 –78, 2000.

[3] K. Bowman, S. Duvall, and J. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *Solid-State Circuits, IEEE Journal of*, 37(2):183–190, 2002.

[4] J. Hu, S. Wang, and S. Ziavras. In-register duplication: Exploiting narrow-width value for improving register file reliability. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 281 –290, june 2006.

[5] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. Irwin. Soft error and energy consumption interactions: A data cache perspective. In *Low Power Electronics and Design, 2004. ISLPED '04. Proceedings of the 2004 International Symposium on*, pages 132–137, Aug 2004.

[6] S. Rehman, F. Kriebel, D. Sun, M. Shafique, and J. Henkel. dtune: Leveraging reliable code generation for adaptive dependability tuning under process variation and aging-induced effects. In *DAC*, pages 1–6, 2014.

[7] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel. Reliable software for unreliable hardware: embedded code generation aiming at reliability. In *CODES+ISSS*, pages pp. 237–246, 2011.

[8] J. Smolens, B. Gold, B. Falsafi, and J. Hoe. Reunion: Complexity-effective multicore redundancy. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 223–234, 2006.

Providing Flexible and Reliable on-Chip Network Communication with Real-Time Constraints

Eberle A. Rambo, and Rolf Ernst
Institute of Computer and Network Engineering
TU Braunschweig, Germany
{rambo|ernst}@ida.ing.tu-bs.de

***Abstract*— The same technology downscaling that enables Multiprocessor Systems-on-Chip (MPSoCs) has increased susceptibility to soft errors, giving rise to the so-called unreliable hardware. In this paper, we discuss the design of a central component of such architectures, the Network-on-Chip (NoC), and how to provide reliable on chip communication for real-time mixed-critical systems, applications with different requirements must be served, regarding e.g. time and reliability. We also discuss formal timing analyses for resilient MPSoCs architectures.**

I. Introduction

Technology scaling increases the hardware susceptibility to soft errors, giving rise to the so-called unreliable hardware. Soft errors are caused by alpha particles from package decay and energetic neutrons from electromagnetic radiation, which are abstracted as bit flips. System design for current and future technologies have to cope with soft errors [1] in order to provide the required reliability levels on each abstraction layer. This further complicates the task of providing response time guarantees, mandatory in the domain of real-time systems.

Techniques to overcome hardware induced errors in software execution can profit from the abundance of cores available in Multiprocessor Systems-on-Chip (MPSoCs) to increase reliability. For instance, software-based fault-tolerance approaches [2], [3] provide reliable execution on a higher level of abstraction without incurring overhead in hardware. Such fault-tolerance approaches assume the correct operation of some key components, both in software and hardware, denominated Reliable Computing Base (RCB) [4].

Figure 1 shows an example of an RCB in a software-based fault-tolerance solution with two types of protection service, replicated execution and checkpointing & rollback. For these approaches to be applicable and, at the same time, financially attractive, the RCB must be as small as possible and must be provided with the lowest overhead. We focus on the hardware part of the RCB, which consists of the inter-core communication, realized by a Network-on-Chip (NoC). Fault-tolerance solutions succeed as long as the communication with the applications works and is reliable. Losing contact with the application instances in a non-predictable manner would mean a system failure. Therefore, the NoC must be highly available and reliable in the presence of soft errors.

In this paper, we discuss the requirements for a resilient Network-on-Chip (NoC) design for use in mixed-critical real-time systems. In this domain, an essential aspect is guaranteeing that the system responds in time, as specified. These guarantees are provided by formal timing analyses, which calculate worst-case response time bounds for the tasks in the system. Since the NoC is a central component and also a heavily shared resource in the system, it plays an important role when providing response time guarantees. Therefore, the NoC must be predictable even under the occurrence of soft errors. Moreover, it has to allow tight worst-case bounds, which rules out many of the available solutions.

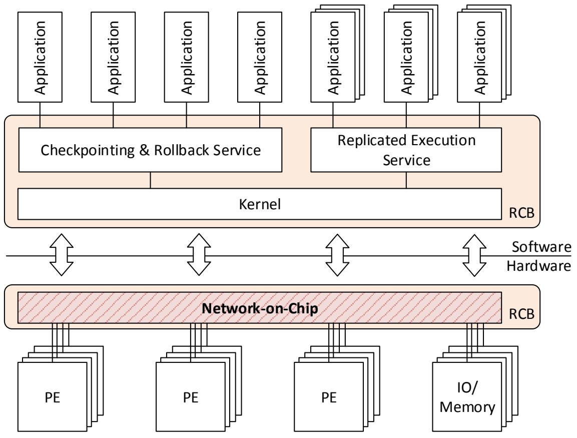


Fig. 1. Example of a Reliable Computing Base (RCB) of a software-based fault-tolerance solution for multiprocessor systems.

II. Reliability vs. Predictability

Many fault-tolerance approaches for increasing the reliability of NoCs have been proposed. The survey in [5] gives a good overview of the existing relevant work. Generally, retransmission protocols, such as Automatic Repeat reQuest (ARQ) [6], are used to correct corrupt data, which is detected using Error-Detecting Codes (EDCs), such as Cyclic Redundancy Check (CRC) [7]. The error detection and correction can be performed on an end-to-end basis, where the traffic is checked at the network interfaces, or on an hop-to-hop basis, where the traffic is also checked at each router. Usually the latter has been preferred. Hybrid schemes between the two have also been researched. Most of the approaches target packet-switched networks. Although faster wormhole-switched NoCs,

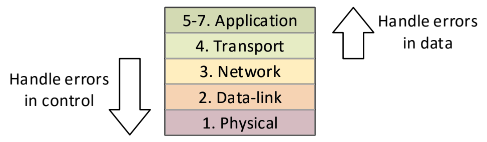


Fig. 2. OSI network model. Errors affecting the control of the network should be handled in the lower network layers. Errors affecting data should be addressed in the upper layers.

where packets are subdivided in Flow Control Units (flits), have also been considered, albeit in the context of general purpose computing.

Despite having been extensively researched, the operation of the router itself under errors has been overlooked. An FMEA analysis of a NoC implementation was performed in [8], [9]. The thorough analysis aims at preparing the NoC for use in real-time safety-critical systems, a domain that requires identifying all possible errors and their effects, which corresponds, in this case, to soft errors and their effects on the NoC routers and links.

The analysis identified several cases where soft errors (transient faults) result in static effects. A static effect caused by soft error means that, for instance, random blocking scenarios could occur in the NoC. The effect of blocking propagates backwards in the network in the form of backpressure and the NoC would only recover with a reset of the affected router(s) or even with a reset of the whole network, both with non-negligible impact on the system performance. Static effects are commonly associated with permanent faults due to similar impacts on the system. If not differentiated at design time, the occurrence of a transient fault with static effects will trigger the recovery mechanism for permanent faults (if implemented) or require a system reset.

Recovery mechanisms for permanent faults consist either of redundancy at design time (Modular Redundancy), mode change or dynamic solutions. One example of dynamic solution is deflective routing, a type of dynamic routing where the traffic is locally redirected to a neighbor router bypassing an unavailable or faulty router. Although well suited for general purpose systems, this class of techniques cannot be applied to real-time systems due to local decision making (dynamic), which drastically impairs the predictability and controllability of the system. Modular redundancy of hardware components (e.g. dual or triple – DMR or TMR) and mode change (or reconfiguration) are expensive features only implemented in systems requiring very high levels of reliability and an extended life. Moreover, those mechanisms have long recovery times, assuming that permanent faults are a rare occurrence. Handling a number of transient faults with these approaches leads to a very long response times in case of errors.

Transient faults should only cause transient effects. We argue that this must be handled in the lower layers of the NoC stack, illustrated in Figure 2. For the sake of reducing area overhead, we discard the use of fault-masking at the routers. In case of errors, corrupt packets are dropped e.g. because the routing data is corrupt and cannot be trusted anymore. This results in a highly available but lossy NoC under soft errors, providing a service comparable to the Internet layer in the TCP/IP protocol stack, where the layer does not guarantee packet delivery but guarantees the routing data integrity [6].

Additionally, the lower layers must provide sufficient independence between communication streams required in a mixed-critical context, which requires that errors do not propagate through different criticalities. This must be provided in the form of fault containment in the lower layers (up to the network layer). As a result of the fault containment, packets are dropped before they are able to affect the traffic from other tasks/criticalities.

On the top layers, transport protocols can provide reliable data transmission. It involves guaranteed packet delivery and guaranteed data integrity services. For instance by using retransmission protocols based on ARQ and CRC as checksum. However, this requires the NoC to be available (i.e. no static effects), otherwise retransmission protocols are ineffective [9]. Moreover, the transport protocol can be easily selected and configured by software in the network interface, according to the tasks' timing constraints and reliability requirements.

III. Integrated Timing Analysis under soft errors

Another essential aspect in the real-time domain is guaranteeing that the system responds in time. This is provided by formal timing analyses, which calculate worst-case response time bounds for the tasks in the system [10]. A simplified illustration is shown in Figure 3. The tasks, which implement functionalities of the system, are mapped to processing resources, the Processing Elements (PEs) in a multi-core. The tasks communicate with other tasks or access resources through the NoC. The analysis must model all relevant aspects of the system in order to provide bounds on the response time of those tasks. The figure abstracts the existence of shared-resources and off-chip communication.

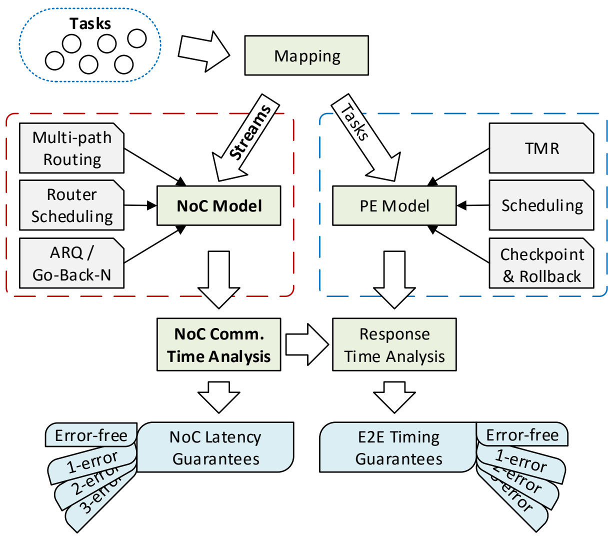


Fig. 3. Formal Timing Analysis flow considering the impact of fault-tolerance mechanisms on the NoC (left-hand side) and on the overall task execution (right-hand side).

The Response Time Analysis provides worst-case response times for the tasks in the system. It is referred to as end-to-end (E2E) response times in Figure 3 because it must enclose and bound all sources of interference, blocking and delays. This includes the scheduling used in each PE, the mapping, the latency to access resources (e.g. DRAM, flash memory) and the latency to communicate with other tasks. In a multi-core, calculating the latency of a communication or resource access is complex, as every communication leaving PE goes through the NoC, a shared resource. These latency bounds are provided by a separate analysis for the on-chip communication.

The Communication Time Analysis of the NoC [10] provides latency bounds for traffic streams on the chip. These can be e.g. cache line transfers, Direct Memory Access (DMA) transfers, sensor value, or a command for an actuator. The analysis must consider the transmission time and all interference that a packet suffers in the network. This is influenced e.g. by the topology of the network, the arbitration in the routers, Quality-of-Service (QoS) mechanisms and traffic classes. Naturally, it also depends on the pattern of the traffic injected by each PE in the system into the NoC.

The integration of fault-tolerance mechanisms and protocols impacts directly these analyses. In addition to the current models, the analysis must account for overheads generated by error detection and recovery, which come in various forms. For instance, ARQ handshaking in the NoC as well as checkpointing in the PEs have impact on the response time even in the error-free case and must be integrated in the respective models.

This causes the need for analyses, such as the ones presented by [10]–[12], to be merged and accurately account for the resulting impacts of errors on the whole system. The worst-case latency and the End-to-End (E2E) response time in different error scenarios, the k-error scenario, can be computed. The analysis of a k-error scenario considers the worst-case impact of k errors on the response time or latency. This tells the system designer whether the system is still able to meet its deadlines in the presence of k errors. The analysis considers that these errors occur inside the busy-window [11]. A realistic k can then be obtained by multiplying the busy-window length ($time$) by the expected error rate ($error \cdot time^{-1}$).

IV. Outline

We have discussed the design of a resilient and reliable Network-on-Chip for real-time mixed-critical systems. The devised solution discards the use of fault-masking in the router for the sake of low hardware overhead. Instead, resilient routers are proposed, where faults are allowed to become errors, whose effects are carefully limited and contained. The resulting network is resilient (i.e. highly available) and predictable under errors, but does not guarantee packet delivery. This is provided by transport protocols on an end-to-end basis. In a mixed-critical system, applications with different requirements must be served, regarding e.g. maximum latency, QoS, safety (freedom from interference). The approach allows flexibility in protecting traffic selectively according to the applications' requirements.

Acknowledgements

This work was partially funded by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 – spp1500.itec.kit.edu).

References

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

[2] P. Axer, R. Ernst, B. Döbel, and H. Härtig, "Designing an analyzable and resilient embedded operating system," in *Proc. on Software-Based Methods for Robust Embedded Systems*, Braunschweig, Germany, 2012.

[3] B. Doebel and H. Hartig, "Can we put concurrency back into redundant multithreading?" in *Embedded Software (EMSOFT), 2014 International Conference on*. IEEE, 2014, pp. 1–10.

[4] M. Engel and B. Döbel, "The reliable computing base-a paradigm for software-based reliability." in *GI-Jahrestagung*, 2012, pp. 480–493.

[5] M. Radetzki, C. Feng, X. Zhao, and A. Jantsch, "Methods for fault tolerance in networks on chip," *ACM Comput. Surv*, vol. 44, 2012.

[6] A. Tanenbaum and D. Wetherall, *Computer Networks*. Pearson Prentice Hall, 2011.

[7] P. Koopman and T. Chakravarty, "Cyclic redundancy code (CRC) polynomial selection for embedded networks," in *Dependable Systems and Networks, 2004 International Conference on*. IEEE, 2004, pp. 145–154.

[8] E. A. Rambo, A. Tschiene, J. Diemer, L. Ahrendts, and R. Ernst, "Failure Analysis of a Network-on-Chip for Real-Time Mixed-Critical Systems," in *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*, 2014.

[9] ——, "FMEA-Based Analysis of a Network-on-Chip for Mixed-Critical Systems," in *NOCS*, 2014.

[10] E. A. Rambo and R. Ernst, "Worst-case communication time analysis of networks-on-chip with shared virtual channels," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15, 2015, pp. 537–542.

[11] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Dobel, and H. Hartig, "Response-time analysis of parallel fork-join workloads with real-time constraints," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*. IEEE, 2013, pp. 215–224.

[12] P. Axer, D. Thiele, and R. Ernst, "Formal timing analysis of automatic repeat request for switched real-time networks," in *SIES*, Pisa, Italy, June 2014.

Reliability and Thermal Challenges in 3D Integrated Embedded Systems

Christian Weis, Matthias Jung and Norbert Wehn
University of Kaiserslautern, Germany

***Abstract*—Tightly coupled embedded system processors and accelerator IPs with memories, such as DRAM or NVM, using 3D integration technology based on TSVs (Through Silicon Vias) offer high bandwidth memory access at low energy consumption. However, those memories, especially DRAMs, are very sensitive to temperature changes since they use capacitors as volatile bit storage elements. The 3D stacking of these layers increases the challenges, such as high power densities and thermal dissipation. It also has a much stronger impact on the retention time of 3D stacked WIDE I/O DRAMs that are mostly placed on top of the heterogeneous embedded processing system (MPSoC). Consequently, it is very important to study the temperature and retention behavior of WIDE I/O DRAMs. Furthermore, there is the need to explore on high-level with advanced holostic modeling how reducing or omitting the refresh in the DRAM influences executed applications. Additionally, it is beneficial to quantify these effects and impact on standard FPGA platforms using commodity DDR3 SO-DIMMs. The outcomes of these investigations permit the transfer of properties and parameters to more advanced 3D stacked systems.**

In this work, we demonstrate the impact of cross-layer DRAM refresh policies on three selected applications ranging from image processing on FPGA, baseband wireless processing with a LDPC-decoder, to a big data application using co-occurrence processing for large graphs. Our results highlight the inherent application resilience wrt. DRAM retention time errors (bit flips) for these very diverging applications.

I. Introduction

The increased power density and thermal dissipation of today's processing and memory systems restrict application performance on smartphones as well as on high-end servers. Advanced fabrication processes that use 3D packaging based on TSV technology enable even tighter integration of systems in a small form factor. These 3D systems start to break down the memory and bandwidth walls. However, this increases largely the power density and reduces the heat dissipation properties of the aggressively thinned dies to enable reliable TSV production. In fact, a 3D stacked SoC aggravates the thermal crisis, which can provoke errors in circuits. This is especially important for DRAMs as they are highly sensitive to temperature changes, and have to be refreshed regularly due to their charge-based bit storage property (capacitor). Thus, advanced error and thermal modeling together with high-level simulators, such as gem5 [1] and DRAMSys [2], are required to investigate current or future processing and main memory systems (based on DDR3 or WIDE I/O DRAMs).

Furthermore, the *retention time* of a DRAM cell is defined as the amount of time that a DRAM cell can safely retain data without being refreshed [3]. This DRAM refresh operation must be issued periodically, and causes both performance degradation and increased energy consumption (almost 50% of future DRAMs total energy), both of which are expected to worsen as the DRAM density increases [4]. Not all DRAM cells behave in the same leaky way, due to process variations. Many prior studies assume that it is possible to keep track of relatively few weak DRAM cells (low retention time) [5], [6] and therefore reducing the impact of refreshing by avoiding the usage of these cells. However, the effects of *data pattern dependence* and *variable retention time* (VRT), which we also observed during measurements, inhibit the application of DRAM retention time profiling mechanisms.

Nonetheless, Liu et al. proposed in [7] an unreliable and a reliable region in the DRAM and refreshed them at different rates. In contrast to [7] we exploit in our work the feasibility of using a *no refresh policy* on three dedicated applications. These three applications are as follows:

- First, we demonstrate the influence of disabling the refresh completely for an image processing task on a XILINX FPGA rotating the image in the DRAM (application specific memory controller).
- Second, we present the input data resilience of a LDPC decoder IP for wireless baseband applications.
- Third and finally, we show for a big data application, namely the co-occurrence calculation based on large data graphs (recommendations for social networks, netflix and more), the impact of the *no refresh policy* on the quality of the prediction.

The results of this work clearly highlight the feasibility of using a no refresh policy on selected applications that can tolerate a certain level of bit error probability (bit flips based on retention time errors in the DRAM). Exploiting this possibility is a viable path for achieving less power consumption and higher available memory bandwidth in a given embedded computing system. The presented investigations are work-in-progress and the results are currently not published.

References

[1] Nathan Binkert, et al. *The gem5 simulator*. SIGARCH Comput. Archit. News, 39(2):1–7, August 2011.

[2] Matthias Jung, et al. *TLM modelling of 3D stacked wide I/O DRAM subsystems: a virtual platform for memory controller design space exploration*. In Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM.

[3] T. Hamamoto, et al. *On the retention time distribution of dynamic random access memory (DRAM)*. Electron Devices, IEEE Transactions on, 45(6):1300–1309, Jun 1998.

[4] Jamie Liu, et al. *RAIDR: Retention-Aware Intelligent DRAM Refresh*. In Proceedings of the 39th Annual International Symposium on Computer Architecture, ISCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[5] J.-H. Ahn, et al. *Adaptive Self Refresh Scheme for Battery Operated High-Density Mobile DRAM Applications*. In ASSCC. IEEE Asian, Nov 2006.

[6] R.K. Venkatesan, et al. *Retention-aware placement in DRAM (RAPID): software methods for quasi-non-volatile DRAM*. In Proc. of HPCA, 2006.

[7] Song Liu, et al. *Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning*. SIGPLAN Not., 46(3):213–224, March 2011.

Improving Code Generation for Software-based Error Detection

Norman A. Rink and Jeronimo Castrillon
Technische Universität Dresden, Dresden, Germany
Center for Advancing Electronics Dresden (cfaed)
Email: {firstname.lastname}@tu-dresden.de

***Abstract*—With modern hardware heading for smaller feature sizes and lower operating voltages, failure rates are expected to increase. Instead of adding hardware to protect against failures, which is expensive and inflexible, one can implement error detection in software. The focus of the present work is the *AN encoding* scheme. Improvements to code generation are introduced that enable a high level of fault coverage, namely over 98%, while reducing the performance overhead due to AN encoding by up to 45%. This raises the general question of whether code generation strategies can be designed that enable fault detection with a minimal impact on performance.**

I. Introduction

In an effort to maintain the exponential performance growth that hardware has been experiencing for the last decades, modern hardware is heading for smaller feature sizes and lower operating voltages. This leads to reduced reliability in both processors [1] and memories [2]. Furthermore, operating *dark silicon* at near-threshold voltage [3] will render computations unreliable. At the same time, the omnipresence of embedded devices in every-day life and industry calls for a high level of reliability, especially in safety-critical applications.

In the future, more resources will have to be spent on protecting against hardware faults. Hardware-based protection is expensive and inflexible. Software-based fault detection and recovery [4], on the other hand, is cheap and can be easily adapted to changing fault scenarios. In a prominent software-based approach the values and operations of a program are *encoded*, and faults are detected by checking whether values are valid code words. To facilitate encoding and checks, extra instructions have to be added to the program, which significantly extends execution time. Longer execution times are justified if unreliable results are turned into reliable ones.

This paper studies the *AN encoding* scheme [5]–[7]. It is shown how the placement of instructions for checking leads to better *fault coverage*, i.e. a higher percentage of detected faults. To this end, we make two improvements to code generation: (i) *improved checks* and (ii) *pinning of checks*. Remarkably, using improved checks also reduces the performance impact of AN encoding.

The structure of this paper is as follows. The concept of AN encoding is introduced in Section II, while Section III describes our specific implementation of AN encoding. Our suggested improvements are assessed in Section IV, and our results are analyzed in Section V. Related work is discussed in Section VI. Section VII concludes our paper.

II. Background and Motivation

Fault detection schemes rely on redundant representations of data: AN encoding uses additional bits to represent encoded values. To encode data, a 32-bit constant A is fixed, and any 32-bit integer value n is replaced with its encoded version $\hat{n} = n \cdot A$. In order to avoid overflow due to encoding, $\hat{n}$ must be represented by 64 bits. In AN encoding, a *check* consists of evaluating the boolean expression

$$\hat{n} \bmod A = 0. \qquad (1)$$

Whenever this evaluates to `False`, a fault is detected. Note that this check requires an expensive modulo operation.

AN encoding is straightforwardly extended to pointers by regarding the address stored in a pointer variable as an integer: an address-valued variable p is replaced with $\hat{p} = p \cdot A$. Checks on $\hat{p}$ are performed in the same way as above[1]. Note that since modern 64-bit systems have 48-bit address buses, no overflow will occur due to AN encoding provided $A < 2^{16}$. In the present work we treat encoding of pointers as an optional extension of AN encoding.

Since AN-encoded programs operate on encoded data, operators must be replaced with encoded versions too. A detailed account of how individual operators are treated can be found in [6]. Here we put particular emphasis on how vulnerabilities arise from certain encoded operations. We denote as $\hat{m}$, $\hat{n}$, $\hat{p}$ the encodings of the values m, n, p respectively. Encoding simple arithmetic operators is straightforward, e.g.

$$\hat{n} +_{\mathrm{enc}} \hat{m} = \hat{n} + \hat{m}.$$

Note that if pointers are encoded, replacement of operators must also be applied to pointer arithmetic. To encode bitwise operators, operands must be decoded:

$$\hat{n} \,\&_{\mathrm{enc}}\, \hat{m} = (n \,\&\, m) \cdot A.$$

Code is vulnerable whenever it operates on the non-encoded values m, n. Memory operations are equally vulnerable, even if pointers are encoded:

$$\mathtt{load}_{\mathrm{enc}}\, \hat{p} = \mathtt{load}\, p.$$

To alleviate the vulnerability due to operating on non-encoded values, encoded values should be checked before decoding. For bitwise operations this leads to the dependency graph in Figure 1a; for memory operations on encoded pointers the

[1]The semantics of the chosen implementation language may require casting p to an integer before encoding it.

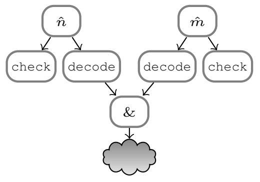


Fig. 1a: Bitwise operation with checks.

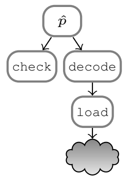


Fig. 1b: Memory operation with checks.

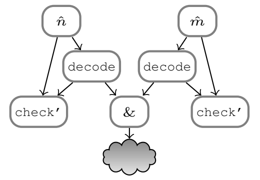


Fig. 2a: Bitwise operation with improved checks.

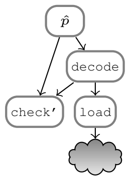


Fig. 2b: Memory operation with improved checks.

graph in Figure 1b is obtained. In both graphs a `check`-node is implemented by evaluating the boolean expression (1).

To understand our first improvement, namely improved checks, note that the `check`-nodes in Figures 1a and 1b are independent of the `decode`-nodes. Thus the compiler may schedule instructions for these nodes far apart, leaving intermediate results vulnerable to faults. Using improved checks, depicted as `check'`-nodes in Figures 2a and 2b, reduces the sizes of vulnerable code sequences: since `check'`-nodes depend on `decode`-nodes, instructions for checking will be placed more tightly around the &-operation and the `load`-instruction respectively. The `check'`-nodes are implemented by evaluating the following boolean expression:

$$n * A = \hat{n}. \tag{2}$$

This implementation also improves performance since it does not rely on an expensive modulo operation. Compilers without sophisticated strength reduction will therefore produce faster code.

Our second improvement is built on a simple observation: Independent of how checks are implemented, a compiler may be able to infer, in certain situations, that $\hat{n}$ is a multiple of A. If this is the case, and based on the optimization level, the compiler may decide to delete instructions for checks. This behavior is undesirable since the observation that $\hat{n}$ is statically a multiple of A ignores the fact that faults occur dynamically. To avoid that checks are optimized away, we can *pin* checks. This is achieved by placing a pseudo-copy instruction around the argument of a check. The result of the pseudo-copy is then fed to the actual check. The pseudo-copy instruction is replaced with a conventional move instruction immediately before register allocation. Thus checks cannot be optimized away, but the register allocator can still coalesce source and target register of the pseudo-copy. This technique was already applied in [8]. Here we analyze its impact on fault coverage and performance.

strategy	type of check	pinning
baseline	(1)	✗
pinned	(1)	✓
improved	(2)	✗
imp., pinned	(2)	✓

TABLE I: Code generation strategies.

III. Implementation

Our implementation of AN encoding is based on the encoding compiler framework that was introduced in [8]. The code transformations that facilitate AN encoding are implemented at the level of LLVM intermediate representation (IR) [9]. However, unlike in [8], only a minimal number of checks is inserted into AN-encoded programs. Specifically, there are only three places where checks are performed:

(a) on the results of a `load`-IR-instruction,
(b) on the non-pointer argument of a `store`-IR-instruction,
(c) on values that are decoded, as explained in Section II.

The checks in (a) ensure that only valid code words enter the program's data-flow. This serves to protect memories, including caches, against hardware faults. If a fault occurs during computations on encoded data, it is highly unlikely that a subsequent fault will turn the corrupted data word back into a valid code word. Therefore the checks in (b) suffice to verify that the final results of computations are valid code words before they are committed to memory. Intermediate checks are not necessary, except where values are decoded, cf. (c). When pointers are encoded, the pointer arguments to `load`- and `store`-instructions must be decoded immediately before these instructions are executed, as in Figures 1b, 2b. Checks are then performed on the pointer arguments due to (c).

Table I gives the definitions of the code generation strategies for AN encoding that are analyzed in this paper. In addition, encoding of pointers can be optionally enabled.

IV. Experimental Setup and Results

The strategies from Table I are applied to the following benchmark algorithms: **Matrix-Vector Multiplication**, **Array Copy**, **Bubblesort**, and **Quicksort**. This set of algorithms represents canonical features of computation, namely arithmetic operations, data movement, and control-flow that cannot be predicted at compile-time. The generated executables are subjected to fault injection experiments and performance measurements, which are conducted on an Intel Core i7 CPU running at 3.6GHz with 32GB RAM.

For fault injection we use Intel's Pin tool [10] together with the BFI plug-in[2]. A single fault is injected into a given test program at run-time as follows. First, one of the instructions executed by the program is chosen at random. Then, a single or multiple random bits are flipped in one of the registers manipulated by the instruction. This fault injection procedure is suitable for simulating transient faults in the combinational logic of a CPU since such a fault will manifest itself in a wrong result being stored in a register.

[2]https://bitbucket.org/db7/bfi

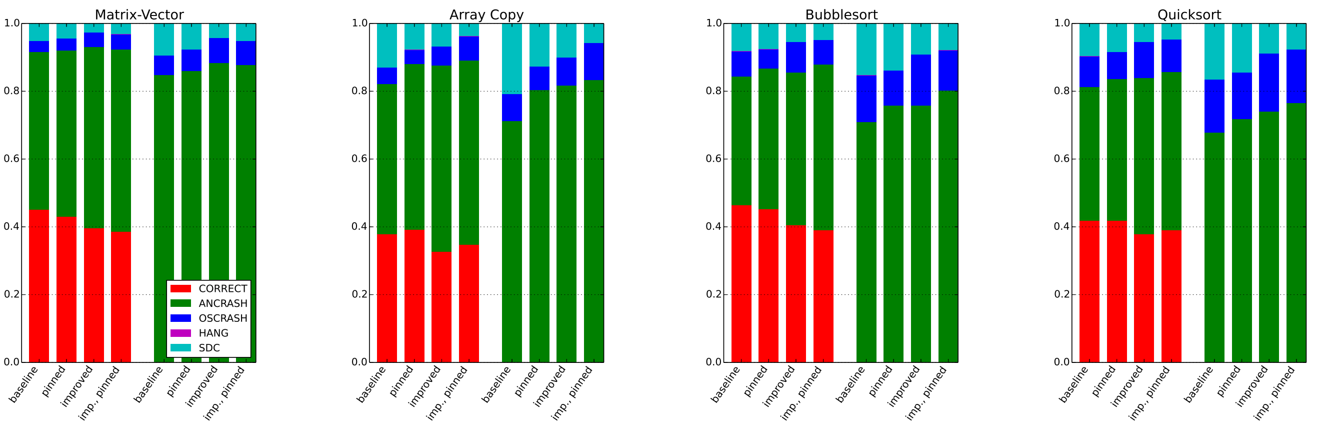


Fig. 3: Fault coverage for encoding of integers only.

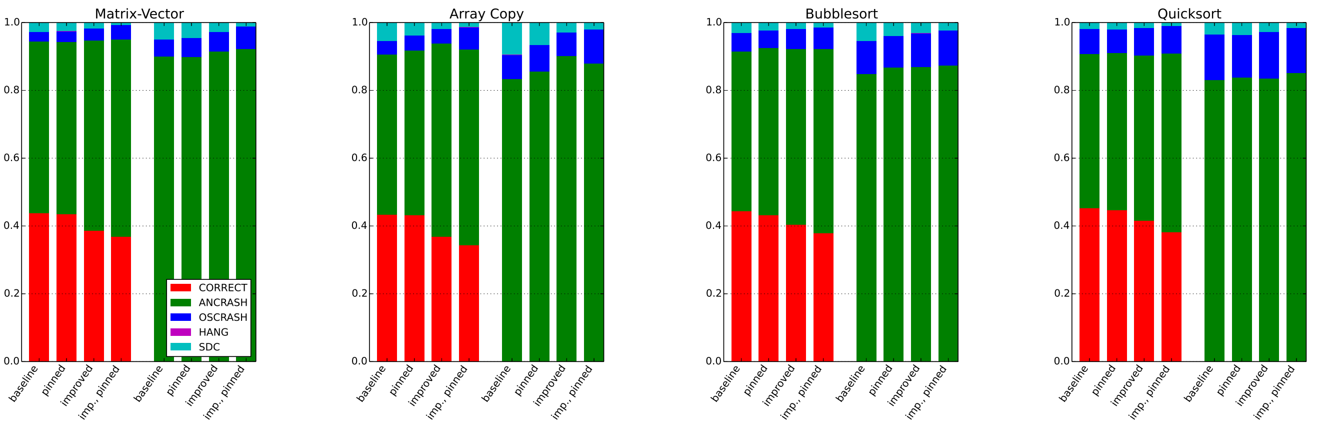


Fig. 4: Fault coverage for encoding of integers and pointers.

For each of our benchmark algorithms combined with each of the code generation strategies we repeat the fault injection procedure 10,000 times. In each of the fault injection experiments program behavior is classified into one of five categories:

1) *CORRECT*: Despite the injected fault the program produced correct output and terminated normally.
2) *ANCRASH*: The injected fault has been detected by one of the inserted checks.
3) *OSCRASH*: The injected fault caused the operating system to terminate the program, e.g. due to a segmentation fault.
4) *HANG*: The injected fault caused the program to take more than $10x$ its usual execution time. The program is therefore deemed to hang.
5) *SDC*: Silent data corruption has occurred, i.e. the program terminated normally but produced incorrect output.

Software-based error detection aims to reduce the frequency of SDC. For the purpose of evaluating our results we therefore formally define *fault coverage* as the frequency of non-SDC results after fault injection. Figures 3 and 4 show the frequencies with which events from the five categories occur. Each of the bars corresponds to one of the strategies from Table I. The sets of bars on the right of each plot are based on the same data as the bars on the left, only the CORRECT events have been left out. We will discuss our findings in detail in Section V.

To measure the performance impact of different code generation strategies, we follow [11] to obtain cycles counts. The number of cycles taken by each executable generated with one of our strategies is divided by the cycles taken when no AN encoding is applied. The resulting quotient is reported as the slow-down due to AN encoding in Table II. The numbers in Table II were obtained for an input array size of 1,000 elements, where each element is a 64-bit word.

V. Discussion of Results

Figures 3 and 4 clearly show that pinning and improving checks both lead to increased fault coverage. If applied individually, improving checks is more beneficial to fault coverage than pinning checks. The highest coverage is generally achieved when both strategies are combined. It is interesting to note that more aggressive checking reduces the CORRECT proportion. This is because faults that do not adversely affect program execution are more likely to be detected if many checks are performed.

	integer encoding only				*integer and pointer encoding*			
	baseline	improved	pinned	imp., pinned	baseline	improved	pinned	imp., pinned
Matrix-Vector	15.72	15.34	18.43	14.12	24.70	23.69	32.29	22.18
Array Copy	7.50	6.07	12.08	8.90	22.30	17.66	29.38	24.63
Bubblesort	3.46	2.77	3.93	3.18	8.08	6.31	9.52	7.21
Quicksort	1.95	1.76	1.98	1.79	3.42	3.00	3.92	3.22

TABLE II: Slow-down due to AN encoding.

	integer encoding only		*integer and pointer encoding*	
	improved over baseline	imp., pinned over pinned	improved over baseline	imp., pinned over pinned
Matrix-Vector	2.5%	30.5%	4.3%	45.6%
Array Copy	23.6%	35.7%	26.3%	19.3%
Bubblesort	24.9%	23.6%	28.1%	32.0%
Quicksort	10.8%	10.6%	14.0%	21.7%

TABLE III: Speed-up due to improved checks.

Our code generation strategies do not only increase overall fault coverage but also the proportion of detected faults, i.e. the proportion of ANCRASH. The single major exception to this rule is the Array Copy benchmark in Figure 4: when improved checks and pinning are combined, the proportion of OSCRASH increases and takes away some portion of ANCRASH. This behavior is caused by faults that affect the program counter, which are responsible for the vast majority of OSCRASH events in the Array Copy benchmark. When improved checks are also pinned, more checks appear in the generated code. Each check is accompanied by a conditional jump instruction to code that should be executed if the check fails. The only register that is modified by the jump instruction is the program counter. This means that more checks in the Array Copy benchmark imply that more faults are injected into the program counter, which, in turn, leads to a greater proportion of OSCRASH.

Figure 3 can be summarized by noting that fault coverage is raised from a low of 87.1% in the baseline strategy to above 95.2% when checks are improved and pinned. When pointers are also encoded, i.e. in Figure 4, coverage is raised from a low of 94.6% to above 98.6%. Protecting pointers is particularly relevant to the observed fault coverage since all of the benchmarks operate on arrays: if a fault causes a bit-flip in the lower bits of an address, it is very likely that the corrupted address is still within the range of the array. Reading from the corrupted address will thus return a valid code word, and hence the fault cannot be detected. However, the computed result will still be incorrect. The effectiveness of encoding pointers comes at a price, as can be seen from Table II. When pointers are encoded, even the slow-downs for the baseline strategy are significantly worse than any of the slow-downs for encoding integers only. The reason for this is that encoded pointers require that every memory access is accompanied by an expensive division operation.

Table II shows that if only integers are encoded, the Matrix-Vector Multiplication benchmark incurs by far the greatest slow-down. This is due to the expensive encoded version of multiplication, cf. [6], [8]. When pointers are also encoded, the slow-down of the Array Copy benchmark is similar to that of Matrix-Vector Multiplication. In other words, the Array Copy benchmark is the one that suffers the worst from pointer encoding. This is unsurprising given that the Array Copy algorithm essentially consists of memory accesses. Table II also proves our claim from Section I that improved checks reduce the slow-down due to AN encoding. For definiteness the speed-ups achieved by using our improved checks are listed in Table III. The best speed-up, namely 45.6%, occurs for the Matrix-Vector Multiplication benchmark.

We conclude this discussion by comparing with previously reported results. The levels of fault coverage achieved for the sorting algorithms in [6] are similar to ours. In [7] fault-coverage for AN encoding is reported between 92%–99%, albeit for a different set of benchmarks. The corresponding slow-downs are in the range of $2x$–$64x$. Variants of AN encoding, namely ANB and ANBD encoding, were also studied in [7]. For these encoding schemes slow-downs of up to more than $256x$ were observed, but a fault coverage of well over 99% is consistently achieved across benchmarks.

VI. Related Work

Following the comparative discussion at the end of the previous section we now give a more general account of previous work on software-based error detection techniques.

AN encoding and its variants, ANB and ANBD encoding, were proposed in [5]. ANB extends AN encoding by assigning a static *signature* to each variable. This enables efficient detection of exchanged operands, which may be the result of bit-flips in addresses, as explained in Section V. ANBD encoding also assigns a dynamic *version* to each variable, and thus detects faults that lead to lost updates. The implementations of AN encoding and its variants in [6], [7] were also based on LLVM [9]. In [6] a detailed account of how operations must be modified in order to operate correctly on encoded values is given. ANB and ANBD encoding achieve fault coverage rates of well over 99% but slow-downs may be as bad as several $100x$. The trade-off between fault coverage and performance in AN encoding was analyzed in [8].

Dual modular redundancy (DMR) detects faults by duplicating instructions and comparing results. To facilitate DMR, automated source-to-source transformations were implemented in [12], which requires compiler optimizations to be disabled in order to ensure that the transformations are not undone by optimization passes. EDDI [13] implements DMR at compiler level, with a focus on instruction scheduling to exploit instruction level parallelism. It was also noted in [13] that the order in which instructions are scheduled can affect the efficiency of detecting faults that lead to invalid control-flow. SWIFT [14] adds control-flow checking to EDDI and also implements simple optimizations at compiler level. ESoftCheck [15] is similar to SWIFT but implements optimizations to remove so-called *non-vital* checks. The fault coverage that is achieved by EDDI is comparable to ours, while SWIFT detects practically all faults. DMR schemes usually lead to slow-downs below $2x$. The advantage of AN encoding over DMR schemes is that permanent hardware faults can also be detected. Moreover, duplication of memory accesses causes issues when DMR schemes are applied on shared memory systems. In AN encoding memory operations are protected without being duplicated.

Δ-encoding [16] merges AN encoding with DMR. Similar ideas were already pursued in [17], [18]. Although the focus of [18] was on fault recovery, it was already acknowledged that scheduling checks close to the uses of values may improve reliability of software-based error detection schemes. Like SWIFT, Δ-encoding also achieves practically full fault coverage. The slow-downs incurred by Δ-encoding are greater than in DMR schemes but generally much lower than for AN encoding. However, Δ-encoding is implemented in [16] as a source-to-source transformation and we believe that it would benefit from improved code generation strategies analogous to our improved checks.

VII. Conclusion and Outlook

Our work has demonstrated that clever implementation decisions can affect instruction scheduling in ways that benefit the quality of software-based error detection. The presented improvements to AN encoding have led to fault coverage of over 98% while reducing the performance overhead by up to 45%. However, slow-downs due to encoding, especially when pointers are encoded, remain large. Our data shows that encoding pointers is crucial to a high level of fault coverage.

In general terms, we have demonstrated that fault coverage can be improved while at the same time lowering the performance overhead. This motivates the development of encoding-specific compiler intrinsics and passes that aid the compiler in generating efficient code that is hardened against hardware faults. Specifically, one could look into ways of giving hints to the compiler that will allow it to reduce the sizes of vulnerable code sequences. Furthermore, understanding the wide variation of speed-ups in Table III might lead to ideas for further improving performance.

Acknowledgments

This work was partially funded by the German Research Council (DFG) through the Collaborative Research Center CRC 912 "Highly-Adaptive Energy-Efficient Computing" (HAEC) and the Cluster of Excellence 'Center for Advancing Electronics Dresden' (cfaed). The authors would like to thank Dmitrii Kuvaiskii and Christof Fetzer for various discussions that helped motivate this work.

References

[1] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs," in *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*. ACM, 2011, pp. 343–356.

[2] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: A large-scale field study," in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '09)*. ACM, 2009, pp. 193–204.

[3] M. B. Taylor, "Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse," in *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. IEEE, 2012, pp. 1131–1136.

[4] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*. Springer, 2006.

[5] P. Forin, "Vital coded microprocessor principles and applications for various transit systems," in *Control, Computers, Communications in Transportation: Selected Papers from the IFAC/IFIP/IFORS Symposium, Paris, France*, 1989, pp. 79–84.

[6] C. Fetzer, U. Schiffel, and M. Süßkraut, "AN-encoding compiler: Building safety-critical systems with commodity hardware," in *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security (SAFECOMP '09)*. Springer, 2009, pp. 283–296.

[7] U. Schiffel, A. Schmitt, M. Süßkraut, and C. Fetzer, "ANB- and ANBDmem-encoding: Detecting hardware errors in software," in *Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security (SAFECOMP '10)*. Springer, 2010, pp. 169–182.

[8] N. A. Rink, D. Kuvaiskii, J. Castrillon, and C. Fetzer, "Compiling for resilience: the performance gap," in *Proceedings of the Mini-Symposium on Energy and Resilience in Parallel Programming (ERPP), Edinburgh, Scotland*, 2015.

[9] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (GCO '04)*. IEEE, 2004, p. 75.

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, 2005, pp. 190–200.

[11] G. Paoloni, "How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures," 2010. [Online]. Available: http://www.intel.de/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf

[12] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano, "A source-to-source compiler for generating dependable software," in *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 2001, pp. 33–42.

[13] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, March 2002.

[14] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*. IEEE, 2005, pp. 243–254.

[15] J. Yu, M. J. Garzarán, and M. Snir, "ESoftCheck: Removal of non-vital checks for fault tolerance," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09)*. IEEE, 2009, pp. 35–46.

[16] D. Kuvaiskii and C. Fetzer, "Δ-encoding: Practical encoded processing," in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015)*. IEEE, 2015.

[17] N. Oh, S. Mitra, and E. J. McCluskey, "ED4I: Error detection by diverse data and duplicated instructions," *IEEE Transactions on Computers*, vol. 51, no. 2, pp. 180–199, February 2002.

[18] J. Chang, G. A. Reis, and D. I. August, "Automatic instruction-level software-only recovery," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2006)*. IEEE, 2006, pp. 83–92.

Resilient System Design through Symbolic Simulation and Online Diagnostics Methods

Thiyagarajan Purusothaman, Carna Radojicic, Christoph Grimm
{thiyag,radojicic,grimm}@cs.uni-kl.de
AG Design of Cyber-Physical Systems
TU Kaiserslautern, Kaiserslautern, Germany

***Abstract*—Safety and dependability have to be considered within the whole life-cycle of a product. In this paper we propose to use (at design-time) model checking methods to verify and get information for (online) self-diagnosis and error reaction. At design-time, we use system models with possible uncertainties in the system and verify safety properties through 'in the loop' simulation. In presence of uncertainties and errors, the reachability analysis is done with safe state definitions. During system in online the compressed representation of reachable states and parameter relations are used to diagnose for errors and execute error reaction functions.**
For this purpose, a vast set of generated diagnostics data within reachability analysis and symbolic simulation of system model is used. In our novel approach, we use Affine arithmetic based modeling of uncertainties and errors in the system, environment and development process. We demonstrate applicability of the approach with a water level controller.

I. Introduction

Due to significance of safety, the standards ISO 26262, IEC-61508, DO-178/254 and CENELEC 50126/128/129 are becoming mandatory part of the design process. For safety, the increasing complexity of today's HW/SW systems is both a blessing and a curse: At one hand, it makes it more likely that components will fail, show unforeseen behavior, or are effected with unexpected scenarios. At the other hand, it allows us to implement error reactions that allow maintaining dependable operation in case of a failure. To analyze safety and fault-tolerance of complex systems, methodologies such as FMEA (Failure Mode and Effects Analysis,) and FTA (Fault Tree Analysis,) are used. FMEA injects single failures in a bottom-up way, and analyzes its impact towards the system. FTA starts top-down with a potential hazard and analyzes possible initiating failures. Despite many limitations [1], [2], both methodologies are still state-of-the art.

With increasing complexity, combinations of multiple errors and deviations are becoming more likely. The complexity of possible dynamic behavior of embedding physical system, HW/SW systems, error models, and the diagnosis- and error reaction systems are tremendous. It demands for exhaustive methods for verification such as model checking. Apart from formal verification during system design, there is often scenarios demanding error reactions with consistency towards formal methods. In particular, for complex systems, the modeling of errors and its interactions requires a more complex understanding. Failing of complex systems is often due to interactions involving different kind of errors:

1) Multiple faults (transient, intermittent or permanent),
2) Inaccuracies, modeling uncertainties or other deviations that accumulate to malfunction,
3) Unforeseen changes or scenarios.

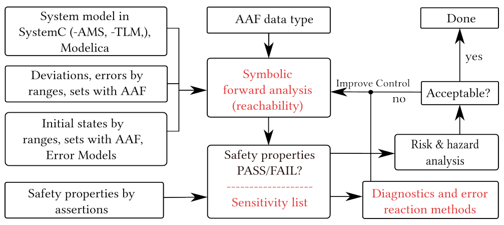
Neither FTA nor FMEA are comprehensive or scale with the complexity and heterogeneity of today's HW/SW systems , as they focus single, maybe two failures with predefined failure modes.

Model checking as a formal method would be comprehensive and allow proving general properties, e.g. safety or liveness properties. General approaches for model checking of mixed discrete/continuous (hybrid) systems are based on linear hybrid automata (LHA, [3]), but don't directly address the presence of failures, uncertainties or unforeseen behavior. Basic idea of model checking of hybrid systems is to use forward and backward analysis (i.e. symbolic simulation) to generate a representation of all possible outputs, on which then properties can be checked in a comprehensive way. An approach that in particular addresses hybrid systems with deviations and uncertainties is the use of Affine Arithmetic [4], which offers an efficient way to compute error (or failure) propagation in a symbolic way. Use of Affine Arithmetic for model checking in presence of deviations of different kind was first proposed in [5].

Towards achieving systems with sufficient resilience, this approach fills the gap in design time verification methods with diagnostics and error reactions. To give systems inherent robustness, designers add on-line diagnostic, error reactions, and even "self-X" properties [6] such as self-healing and self-organization to it's basic functionality. A first use of model checking for detecting erroneous (=unsafe) behavior was proposed in [7] in an autonomous driving case study: model checking was used "online" to compute whether unsafe states are reachable. With the basis from symbolic simulation in [5], we extend our work [8], [9] to model and compute sensitivity for each HW/SW component at system level. This allows us modeling of potential errors. Main contribution of this work is the use of the information generated by this model checking method for implementing diagnostics and error reaction functions. Compared with "self-X" methods and [7], we reduce the computational overhead by using off-line generated information that speeds up on-line diagnosis and allows to compute error reactions.

II. Offline Affine simulation and Reachability analysis

The figure 1 gives an overview of the approach, applicable to a simplified development process. We assume a model-based development: the HW/SW system including its embedding physical environment are modeled using SystemC (-AMS, -TLM) and Modelica. Potential errors and deviations of different kind are modeled by ranges or symbols that specify ‘unknown’ or arbitrary behavior. To cover multiple, may be all possible scenarios, the starting states and inputs can be ranges (continuous quantities) or ‘unknown’ states (discrete variables). We implement ranges and unknown states by using an abstract data type AAF. This abstract data type implements symbolic operations and provides overloaded operators and functions. Symbolic forward analysis, a first step of model



System model in SystemC (-AMS, -TLM,), Modelica
Deviations, errors by ranges, sets with AAF
Initial states by ranges, sets with AAF, Error Models
Safety properties by assertions
AAF data type
Symbolic forward analysis (reachability)
Safety properties PASS/FAIL?
Sensitivity list
Improve Control
no
Done
yes
Acceptable?
Risk & hazard analysis
Diagnostics and error reaction methods

Fig. 1. Overview of methodology.

checking, then computes for the HW/SW system in its environment, considering all combinations of given errors/deviations and tolerances in different components. After this step, the fulfillment of safety properties that are specified by assertions is checked. Through this affine based symbolic simulation the violation of safety properties are evaluated formally. The method also defines the safe and unsafe state with sensitivity list and captured into AAF data type. This list can be used

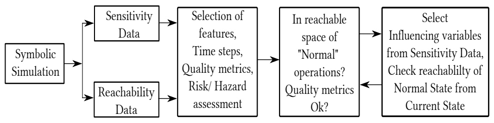
1) For a more detailed analysis of risks and hazards, considering also probabilities of errors.
2) Together with reachability analysis, we use it for development of diagnostics and error reaction methods.

A. Online Diagnostics and Error Reaction

In figure 2 shows the online steps in the system to diagnose for potential errors. The potential errors are due to combination of errors, variations and deviations at system level. The online diagnostics is precise based on the offline simulation. The identification of simulated region happens by parameter identification through sensitivity list and system state for each component. The sensitivity and reachability data guides the online resilience software to diagnose for errors and error reaction is triggered (ref Fig 2). With the influence of offline simulation data, the influencing variables or parameters are calibrated to achieve safe state.

III. Outlook

In our approach, we have introduced affine based simulation for formal representation of errors, tolerances and unexpected scenarios. This simulation approach helps us to formally define system safe state space and recommends resilience methods for violations of safety properties. Through which the approach also derives the sensitivity list for the



Symbolic Simulation
Sensitivity Data
Reachability Data
Selection of features, Time steps, Quality metrics, Risk/ Hazard assessment
In reachable space of "Normal" operations? Quality metrics Ok?
Select Influencing variables from Sensitivity Data, Check reachablilty of Normal State from Current State

Fig. 2. Symbolic Simulation Data and Online usage

safe and unsafe states. The figure 3 depicts the test set-up for a water level controller with our resilience approach. The right hand figure explains the diagnostics reaction due to level sensor failure. The red-color is the upper and lower water limit and blue color is due to effect of resilient software on sensor failure (Fig. 3 right). When this system is in a real-time scenario, with the parameter relations and pre-simulated sensitivity list, the diagnostics reactions are executed. Based on this current state of work, we would further develop towards the fault tree construction and analysis with automatic multi-layered diagnostics.

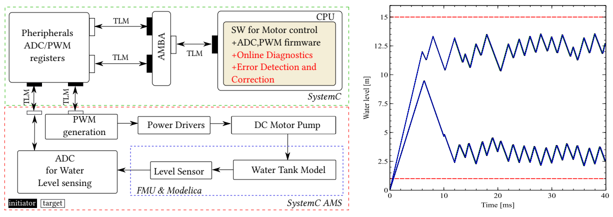


Fig. 3. Experimental setup and results for resiliency

Acknowledgment

This a part of the ANCONA project, this work has been supported by BMBF (Bundesministerium für Bildung und Forschung) program IKT 2020 under contract no. 16ES021.

References

[1] N. Shebl, B. Franklin, and N. Barber, “Is failure mode and effect analysis reliable?” *Journal on Patient Safety*, vol. 5, pp. 86–94, 2009.

[2] N. Bidokhti, “FMEA is not enough,” in *Reliability and Maintainability Symposium, 2009. RAMS 2009. Annual*, Jan 2009, pp. 333–337.

[3] T. A. Henzinger, “The theory of hybrid automata.” IEEE Computer Society Press, 1996, pp. 278–292.

[4] M. Andrade, J. Comba, and J. Stolfi, “Affine Arithmetic (Extended Abstract),” *Interval ’94, St.Petersburg, Russia*, 1994.

[5] C. Grimm, W. Heupke, and K. Waldschmidt, “Analysis of mixed-signal systems with affine arithmetic,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, no. 1, pp. 118–123, 2005.

[6] C. Müller-Schloer, H. Schmeck, and T. Ungerer, Eds., *Organic Computing - A Paradigm Shift for Complex Systems*. Birkhäuser.

[7] M. Althoff and J. Dolan, “Online Verification of Automated Road Vehicles Using Reachability Analysis,” *Robotics, IEEE Transactions on*, vol. 30, no. 4, pp. 903–918, Aug 2014.

[8] C. Radojicic, C. Grimm, J. Moreno, and X. Pan, “Semi-Semi-Symbolic Analysis of Mixed-Signal Systems including Discontinuities,” in *Proccedings of Design, Automation and Test in Europe 2014 (DATE ’14)*, 2014.

[9] C. Radojicic, T. Purusothaman, and C. Grimm, “Towards formal validation: Symbolic simulation of systemc models,” in *Proceedings of International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS 2015*, 2015.

Checkpointing virtualized mixed-critical embedded systems

Aitzan Sari
Dept. of Informatics
University of Piraeus
Piraeus, Greece
aitsar@unipi.gr

Mihalis Psarakis
Dept. of Informatics
University of Piraeus
Piraeus, Greece
mpsarak@unipi.gr

***Abstract*— Last years the virtualization technology is gaining acceptance in embedded systems world since it provides the means for improving reliability, security and isolation of applications in a mixed-critical embedded environment. In this work, we study various implementation issues of a checkpointing mechanism in a virtualized embedded system running mixed-critical applications. We port the widely-used, open-source Xen hypervisor in an ARM-based embedded platform and install different guest operating systems atop of it. We implement various checkpointing schemes for our virtualized embedded system depending on the different reliability requirements of the virtual machines and hosted applications. Finally, we measure the impact of the embedded hypervisor in the checkpointing process and consequently in system performance.**

Keywords—checkpointing; virtualization; Xen hypervisor; mixed-critical embedded systems

I. Introduction

Nowadays, virtualization, which enables the hosting of multiple virtual machines (VMs) in a single hardware platform, is widespread in server and desktop computing domains. Virtualization allows a single physical server to act as multiple, separate logical servers each one running its own operating systems, thus providing isolated environments but with high resource utilization. On the other hand, desktop virtualization is mainly used to allow the users to load a second OS in order to run applications not supported by their primary OS. A recent trend both in academia and industry suggests the use of virtualization in embedded systems [1], [2].

The motivation of building virtualized embedded systems is different from that of building virtualized servers: in servers similar virtual machines (and operating systems) run to serve multiple users having similar computing needs, whereas in embedded systems the virtualization is mainly adopted to handle heterogeneity and host applications and operating systems with different features in the same hardware platform. For example, in automotive domain, virtualization enables the integration of two different computing worlds in the same hardware platform [1]: typical control/convenience car functions running in an automotive real-time operating system, e.g. AUTOSAR and infotainment functions running in a popular operating system, e.g. Linux or Windows can co-exist in a virtualized embedded system. Therefore, the traditional techniques and solutions used in virtualized servers must be revisited and adapted for the case of embedded systems. To this end, recent approaches have studied and evaluated existing virtualization technologies in real-time embedded systems [3], [4] and have proposed real-time versions of popular virtual machine monitors, like Xen [5], [6].

Among the advantages of virtualization technology is its proactive fault tolerant mechanisms through the use of VM checkpointing and migration [7]. VM checkpointing is the process of save and restore the VM state. VM checkpointing is useful for tolerating runtime errors and increasing system availability but also for migrating a VM to another machine. Several checkpointing techniques have been presented in the literature, from the first implementation of a virtual machine checkpoint/restart mechanism based on Xen hypervisor [7] to more recent approaches [8], [9], [10]. In [7], authors describe a fault tolerant virtualized system based on the checkpoint/restart approach while presenting an infrastructure for the checkpoints' management. The work in [8] provides an efficient method for the reduction of the required checkpoint memory space by storing VM's data that have been modified recently. The proposed method keeps track of the guest's IO operations to the external storage device and maintains a list of duplicated memory pages which are excluded at the checkpoint operation. The authors in [9] analyze the performance overhead of system calls for application-level checkpoints and propose a hypervisor-assisted model which supports the tracking of faulty memory pages and allows the applications to directly use the hypervisor primitive functions. Finally, in [10], VM-μCheckpoint is implemented for high-frequency checkpointing and recovery of VMs. Its goal is to minimize checkpoint overhead and to speedup system restore while also considers the issue of corrupted snapshots due to fault detection latency in high-frequency checkpoints.

The more recent checkpoint/restore approaches are aiming to minimize performance overhead and reduce checkpoints (snapshots) size. However, the application of checkpointing has not been studied for the case of virtualized embedded systems and the impact of virtualization in the functions of a checkpointing mechanism has not been analyzed. Furthermore, there are several implementation issues that must be considered and evaluated, for example where snapshots are saved, how the different criticality features of VMs in a mixed-critical system and the real-time constraints of the RTOS-based partitions affect the scheduling and frequency of checkpointing.

In this paper, we port the open-source Xen hypervisor [11] in an ARM-based platform, Cubieboard2 [12]. Cubieboard2 integrates an Allwinner A20 SoC that features a dual-core ARM Cortex-A7 CPU. Note that ARM Cortex-A7 implements ARMv7 architecture which includes ARM extensions for virtualization. We setup a mixed-critical virtualized embedded system with three partitions and different guest operating systems (two Linux OS and a FreeRTOS) assuming benchmarks of different criticality and implement a checkpointing mechanism for it. For the checkpointing of the Linux partitions we adopt the Berkeley Lab Checkpoint/Restart (BLCR) library while for the FreeRTOS we implement our checkpointing function. Our main target is to investigate the impact of the virtualization environment to the checkpointing/restore procedure while also exploring several checkpointing implementation issues. To this end, we present three checkpointing/restore schemes: an application level scheme, an application-system level and a system level scheme. In the application level, the checkpointing/restore is managed and executed by the guest VM while in the application-system level the entire process is coordinated by the privileged VM and executed by the guest VM. These schemes are application-based meaning that the checkpoint/restore is executed each time in a single application running in a guest VM as opposed to the system level where the checkpoint/restore is executed in the entire guest VM. For the application-system and the system level checkpointing, we implemented a checkpointing scheduler/controller running on the privileged domain (dom0) which communicates with the guest OSs to suspend the VM, saves its state and resumes it. We calculate the impact of the virtualization in the checkpointing procedure for different embedded benchmarks.

II. Checkpointing Scheme

Several issues are arisen considering checkpointing/restore in mixed-critical embedded systems operating in virtualized environment:

a) *Snapshots storage:* Traditional server checkpointing mechanisms save snapshots to a non-volatile storage medium (e.g. disk), not possible in most embedded systems. A checkpointing approach for virtualized embedded systems must consider the storage of VM snapshots in a shared volatile memory while preserving VM isolation and security.

b) *Criticality level:* Checkpointing mechanisms proposed so far for virtualized servers assume that all virtual machines have the same reliability/availability requirements, and thus they apply a common checkpointing scheme to all VMs. On the contrary, in a mixed-critical virtualized embedded system where VMs have different reliability requirements, a non-uniform checkpointing scheme should apply. For example, in a high-critical partition (i.e. a VM running safe-critical applications) checkpointing frequency should be higher compared to a medium or low-critical partition.

c) *Checkpointing frequency:* To determine the checkpointing frequency in a virtualized embedded system, one must take into consideration not only the criticality levels of the partitions but also the other system constraints, such real-time or power consumption constraints.

To evaluate the impact of such issues in the reliability and performance we must develop a checkpointing mechanism for a virtualized embedded system and run various experiments. In this work, we develop three checkpointing/restore schemes for a mixed-critical virtualized embedded system. The proposed approaches are based on the criticality level of the running applications and the criticality level of the guest VMs. The snapshots are stored in isolated volatile memory area to reduce the memory access time for the save/restore snapshots. The entire process is coordinated by the guest VM or the hypervisor depending of the checkpointing level. As mentioned earlier, in the *application level*, checkpointing is managed by the guest VM while in the *application-system* and *system level* the process is coordinated by the privileged domain. *Application level* and *application-system level* checkpointing aims at providing fault tolerance to the application level when a fault affects specific application(s) while the VM is able to run without interruption. In this scenario, only the affected application is needed to be restored reducing thus the fault recovery time. In case an error affects the execution of a VM, all running application are considered as faulty and complete VM restore is required to resume its fault-free execution. For this reason, *system-level* checkpointing is required to tolerate faults affecting the guest VMs. In order to take *application-level* snapshots, a specific application is running in each VM managing the checkpointing process which is implemented depending on the applications' criticality of the particular VM. In the *application-system level* the same specific application is executed in the VM but for the management of the checkpointing, it listens to incoming messages from the hypervisor to save and restore applications as directed by the received messages. This thread (checkpointing/restore thread) resides inside each guest and is executed with the highest priority to be able to take a process checkpoint as soon as directed by the hypervisor and to restore an application as soon as a fault is detected. In the *system level* checkpointing, the scheduling application running in the privileged domain takes care of the checkpointing process which is predefined according to the system VMs reliability requirements. In contrast to the *application level* checkpointing where a fraction of the VM's memory region is copied and restored, in the *system level* checkpointing, a snapshot of the entire VM's memory is taken and stored in predefined memory area.

At this point it should be helpful to provide some basic information about the Xen Hypervisor. Xen is an open-source Type-1 hypervisor (hypervisor running directly on the system hardware) and can run many instances of different operating systems (called guests or domains). A special privileged domain (Dom0) is required which contains the device drivers and tools to control (create, delete, configure) the other domains of the system and is the first VM executed on the system. The other VMs are called DomU (unprivileged domains) since they are completely isolated from the hardware

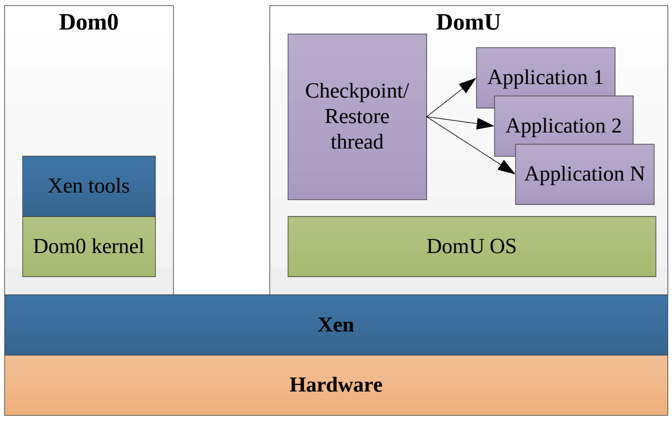


Fig. 1 Application level checkpoint/restore scheme

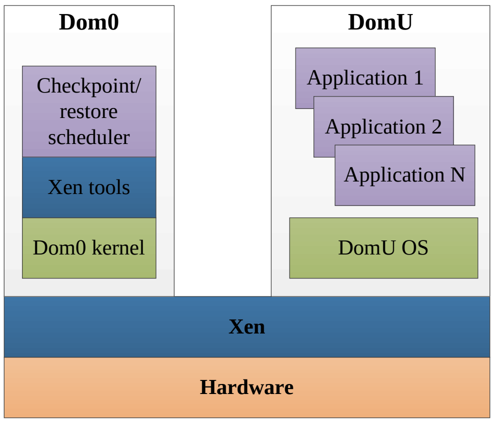


Fig. 3 System level checkpoint/restore scheme

with no privileges to access hardware resources directly. Guest OSs could be HVM (full or hardware-assisted virtualized) or PV (Paravirtualized) and can be used at the same time with the hypervisor. In this work we are considering full virtualized guests.

Fig. 1 depicts the *application level* checkpointing where the process of saving and restoring applications is managed and executed by the VM itself depending on real-time constraints of the running applications. Fig. 2 illustrates the *application-system level* checkpointing/restore scheme. In this concept, the privileged VM (Dom0) runs the *Checkpoint/Restore scheduler* which coordinates the application of the checkpointing mechanism in the entire system. Which VM and which process should be checkpointed more frequently is system-dependent; in this work the criticality and the reliability requirements of the VMs and their applications are considered as criterion to the checkpointing scheduling and frequency (i.e. applications with higher criticality have higher checkpointing frequency). Moreover, as mentioned earlier, each unprivileged VM (DomU) runs a checkpointing/restore thread which listens to incoming messages from Dom0 and takes checkpoints of a given application or restores a fault-free snapshot. Both these approaches allow fault isolation between different processes. This means that while a specific process of a VM has been marked as erroneous and is restored, its other processes can continue running uninterruptedly. Its main disadvantage is that

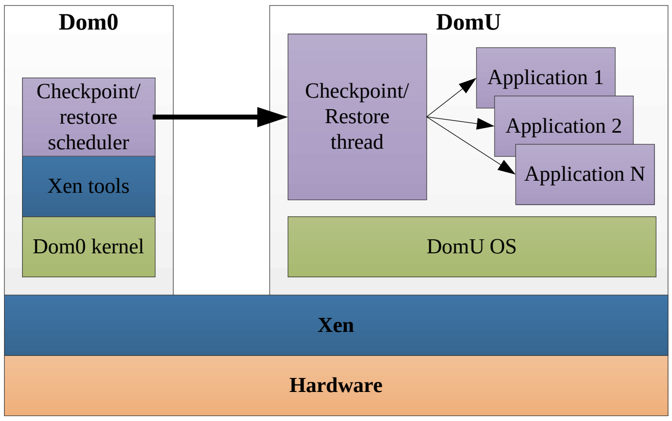


Fig. 2 Application-system level checkpoint/restore scheme

since the checkpoint snapshot resides inside the memory region of the guest VM, in case a fault propagates to the VM and affects its execution state, the applications would not be able to restore.

The *system level* checkpointing/restore approach is depicted in Fig. 3. In this approach, the Checkpointing/restore scheduler running in Dom0 is responsible to take checkpoints of the guest-VMs. This is achieved by copying the entire memory region of the guest VM in a predefined memory area. Upon fault occurrence, the guest VM is halted, the taken VM snapshot is copied back to the original memory area and finally the VM is resumed. As with the application-level checkpointing, the scheduling and frequency of the checkpoints is based on the criticality levels of the VMs. This requires system partitioning in terms of guest-level criticality, meaning that the applications are assigned to the guest VMs according to their criticality levels and checkpoints are taken more often to the VM with the higher criticality level. As opposed to process-level checkpointing, the VM snapshots reside outside the memory region of the guest VM thus providing a more robust solution for faults affecting the state of the VM.

III. Experimental Results

In order to evaluate the proposed checkpointing/restore schemes we have ported the Xen hypervisor on an ARM Cortex-A7 CPU embedded platform along with three guest-OSs: two Linux guest VMs and a FreeRTOS VM. We used the MiBench benchmarks [13] as system applications and particularly, four benchmarks from the automotive suite (basicmath, bitcount, qsort and susan) and the FFT algorithm from the communication benchmarks. For the checkpointing/ restore mechanism in the Linux guests we have adopted the Berkeley Lab Checkpoint/Restart (BLCR) [14] application while a custom thread-level checkpoint/restart application is implemented for the FreeRTOS VM (Fig. 4).

Table 1 shows the snapshot sizes and the execution time of the checkpointing process for all benchmarks in the *application level* scheme. Table 1 also shows the execution times of the checkpointing process in native OS in order to measure the overhead imposed by the virtualization environment. The results are taken executing the benchmarks many times and are

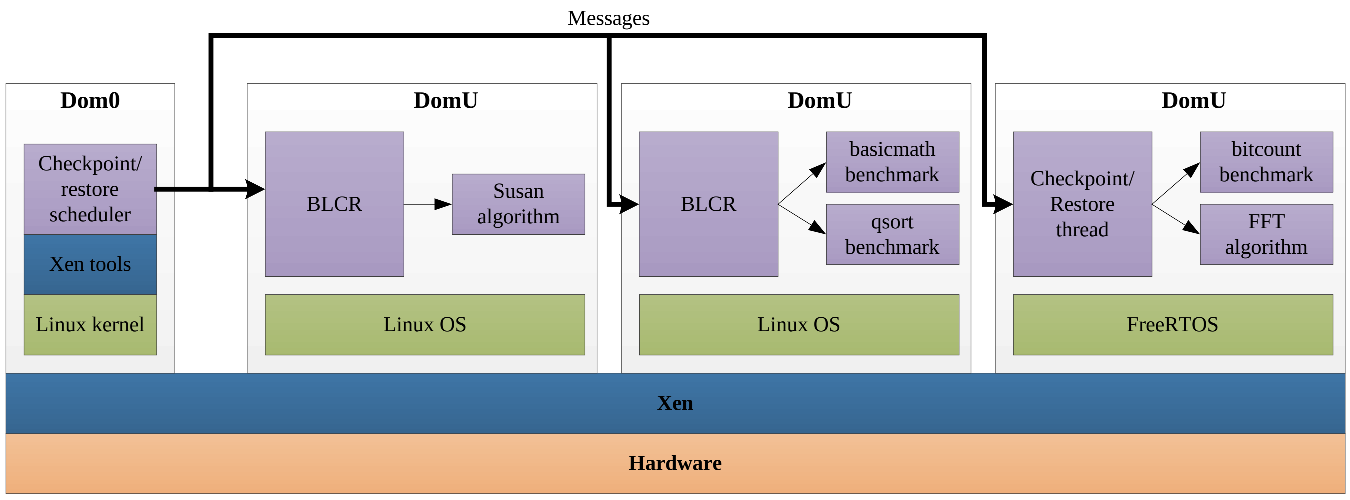


Fig. 4 Implemented virtualized embedded system

given here as the mean time of these executions. As we can see the overhead imposed due to virtualization varies between 13.17% in the Qsort application and 40.45% in the basicmath benchmark with mean time overhead of 24.70%. In applications with real-time constraints this overhead should be considered to the system schedulability in order the applications to be executed within their deadlines. Notice that the overhead increases, but not analogously, with the memory size.

TABLE 1. APPLICATION LEVEL CHECKPOINTING

Benchmark	Snapshot size (KBytes)	Native OS checkpoint time (msec)	App. level checkpoint time (msec)	Virtualization overhead (msec)
FFT	17	2.07	2.40	0.33
bitcount	21	2.37	3.22	0.85
basicmath	9	1.78	2.50	0.72
Qsort	42	8.43	9.54	1.11
Susan	30	6.91	8.16	1.25

For the application-system level, the same experiments as the application level have been done coordinated by Dom0 scheduler. In this case we did not observe additional overhead compared to application level scheme. This is because Xen messages between guests do not impose significant overhead. Table 2 shows the system level experiments, where we assigned 256MB RAM for the Linux guests and 128MB for the FreeRTOS guest. Taking memory snapshots for the system guests, using Xen built-in utilities, took 960msec for the Linux guests and 502msec for the FreeRTOS guest.

TABLE 2. SYSTEM LEVEL CHECKPOINTING

Virtual Machine	OS	Memory (MB)	Save time (msec)
DomU-1	Linux	256	960
DomU-2	Linux	256	960
DomU-3	FreeRTOS	128	502

IV. CONCLUSION AND FUTURE WORK

In this paper, we analyzed the impact of the virtualization environment in fault tolerant embedded systems where checkpointing/restore is used for fault recovery. We used Xen hypervisor on ARM Cortex-A7 CPU with three guest domains (two Linux and a FreeRTOS) and study different checkpointing schemes and their impact in the checkpointing time. The results showed that although virtualization can be used to improve reliability by isolating the guest domains, it adds significant time which should be considered in checkpointing/restore fault recovery schemes.

As a future work, we aim to apply and analyze the impact of the checkpointing frequency in system availability and implement different checkpointing algorithms mostly used in RTOSs for task scheduling.

ACKNOWLEDGMENT

This research was co-financed by the European Union and Greek national funds through the projects “Validation of Time and Space Partitioning (TSP) architectures in emerging space-craft/satellite on-board applications” (Bilateral R&D Cooperation Program between Greece and China 2012-2014/ VISPA) and “Hardware and Software Techniques for Multi/Many-core Processor Architectures Reliability Enhancement (Thalis/HOLISTIC)”.

REFERENCES

[1] G. Heiser, "Virtualizing embedded systems - why bother?," in Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE, 2011, pp. 901-905.

[2] A. Aguiar and F. Hessel, "Embedded systems' virtualization: The next challenge?," in Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on, 2010, pp. 1-7.

[3] M. Aichouch, J. C. Prevotet, and F. Nouvel, "Evaluation of the overheads and latencies of a virtualized RTOS," in Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on, 2013, pp. 81-84.

[4] K. Sandstrom, A. Vulgarakis, M. Lindgren, and T. Nolte, "Virtualization technologies in embedded real-time systems," in Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on, 2013, pp. 1-8.

[5] X. Sisu, J. Wilson, L. Chenyang, and C. Gill, "RT-Xen: Towards real-time hypervisor scheduling in Xen," in Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on, 2011, pp. 39-48.

[6] Y. Seehwan and C. Yoo, "Real-Time Scheduling for Xen-ARM Virtual Machines," Mobile Computing, IEEE Transactions on, vol. 13, pp. 1857-1867, 2014.

[7] G. Vall´ee, T. Naughton, H. Ong, and S. L. Scott, "Checkpoint/restart of virtual machines based on Xen," in High Availability and Performace Computing Workshop (HAPCW 2006), Santa Fe, New Mexico, USA, 2006.

[8] E. Park, B. Egger, and J. Lee, "Fast and space-efficient virtual machine checkpointing," SIGPLAN Not., vol. 46, pp. 75-86, 2011.

[9] L. Min, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik, "Hypervisor-assisted application checkpointing in virtualized environments," in Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on, 2011, pp. 371-382.

[10] W. Long, Z. Kalbarczyk, R. K. Iyer, and A. Iyengar, "VM-μCheckpoint: Design, Modeling, and Assessment of Lightweight In-Memory VM Checkpointing," Dependable and Secure Computing, IEEE Transactions on, vol. 12, pp. 243-255, 2015.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, et al., "Xen and the art of virtualization," presented at the Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing, NY, USA, 2003.

[12] http://cubieboard.org/.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on, 2001, pp. 3-14.

[14] Duell J., Hargrove P., and Roman E., "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart," Berkeley Lab Technical Report (publication LBNL-54941), 2002.

Methods of Timing Reliability Improvement for Combinational Blocks in Microelectronics Systems

Sergey Gavrilov
Institute for Design Problems in Microelectronics
of the Russian Academy of Science (IPPM RAS)
Moscow, Zelenograd, Russia
s00v@yandex.ru

Galina Ivanova
Institute for Design Problems in Microelectronics
of the Russian Academy of Science (IPPM RAS)
Moscow, Zelenograd, Russia
pirutina_g@ippm.ru

***Abstract* — One of the critical factors of any combinational block resiliency is timing reliability. For reliable chip design, it is necessary to predict the delay variation due to different factors, such as uncertainty of technological and circuit parameters, internal cross coupling noise, external noise and others. The most critical timing value is the worth case or maximal delay. But there are many issues where minimal delay is also important, for example, trigger hold constraint satisfaction, switching activity for peak current or IR-drop analysis, timing windows intersection analysis for noise masking, etc. So, the accurate timing window analysis with minimal and maximal delays estimation is important for many applications.**

In this paper we propose an approach for timing windows propagation for different input stimulus simultaneously. To increase the reliability of timing windows analysis, we try to unite two opposite approaches for delay estimation, namely, logic simulation for a set of particular input stimulus and static timing analysis. This combination gives the ability to reach the reasonable compromise between speed and accuracy accounting for internal logic analysis and false path rejection.

Our contributions into timing windows analysis are the new interval characteristic function (ICF) mechanism for logic compatibility analysis of different timing windows and accurate model with characterization of the correctional difference between delay element without and with considering simultaneous inputs switching.

Keywords* — *logic simulation; static timing analysis (STA); interval simulation; binary decision diagram (BDD)

I. Introduction

Normally two opposite approaches are used for timing analysis. The first approach is the electrical or logic simulation for a set of particular input stimulus. This approach is the most accurate and it and takes in into account internal logic of analyzed combinational block. But, this approach can not be exhaustive for a large number of primary inputs, because the number of possible input stimulus is increased exponentially with increasing primary input number.

The opposite approach is the static timing analysis (STA). The purpose of the STA is that to find the set of critical paths in combinational blocks for subsequent evaluation of circuit period and frequency. STA is the real opportunity to solve the problem of exhaustive timing analysis, but the STA solution is often too pessimistic with overestimation of timing windows, because the normal STA does not take into account the internal logic of the circuit. Currently, the most widely used tools for the analysis of critical paths are Synopsys Prime Time and Cadence CTE Encounter. There are a lot of papers and approaches to solve the problem of the truth or the falsehood of the critical path. But these approaches are not widely used in commercial systems since the problem itself is NP-complete, and there is no efficient algorithms for circuits with large dimensions.

In this paper we try to combine two opposite approaches for delay estimation, namely, logic simulation for a set of particular input stimulus and static timing analysis. We consider problems of complex digital circuit performance analysis with the uncertainty of technological and circuit parameters. The traditional performance analysis of test stimulus sequence orders events during time, while the proposed technique provides space ordering.

A wide set of digital circuit simulation problems requires both maximal node delay and minimal delay. The accurate minimal delay model depends on glitches and simultaneous gate input switching. But, the existing logic level performance analysis tools, as a rule, use simplified pin-to-pin gate delay model. This paper describes the method, which provides considerable logic level interval delay analysis accuracy versus the famous approaches for simultaneous multiple input switching [1-2].

II. Related Works

In order to raise the authenticity of the static timing analysis results, attempts were repeatedly made to take into account the logic of the circuit operation in critical path analysis. In particular, express search methods of logical correlations inside the circuit and their application for eliminating false paths were aimed to solve this problem [3-9]. Undoubtedly, this approach substantially increases the accuracy of the performance estimation compared to the classical static timing analysis; however, it does not assure achievement of exact limits of the possible delay range.

The true path search technique, based on the recursive construction of the path sensitization function from the preset input action, solves the contrary problem [10]. The true value of the path sensitization function ensures the presence of at least one input vector that determines the delay equal to the length of this path. However, the opposite is not true, and, when the sensitization function value is false, the path can prove to be true. In order to reduce the uncertainty in the analysis of the truth of the path, the inverse functions (co-sensitization) were proposed [10]. Such functions assure the path's falsity for the true function value. Nevertheless, in this case, the completeness of the analysis is not ensured and there are situations when sensitization is true and co-sensitization is false, and it is impossible to conclude from these function values on the truth or falsity of the path.

In order to ensure the completeness of the delay analysis on exposure to different input effects, the so-called arithmetic decision diagrams (ADDs) were proposed [11]. The ADDs, as well as the classical binary decision diagrams (BDDs) [12], use binary input action vectors as arguments. However, the value of the arithmetic delay, but not the value of the Boolean function is determined at the output in the leaf nodes of the ADD. This approach describes the complete delay spectrum, including the range limits. It is known that this method is not applicable for large circuits due to the large memory consumptions and is efficient only for the rough rounding of delay values up to integer values.

Timing and logic correlations have been used very successfully to reduce pessimism in functional noise analysis where a noise glitch is responsible for flipping the logic state of a quiet victim net. Timing correlations are computed by propagating the switching windows obtained through static timing analysis [13-15]. This technique is relatively simple and effective. However, this approach does not identify situations where all aggressor nets can not switch at the same time in the same direction due to logical constraints in the circuit. In order to eliminate all false noise failures, both timing and logic correlations of the signals must be taken into account. In [14], this problem was represented as a search for a worst-case 2-vector test using a Boolean Constraint Optimization formulation. A test pattern generation approach was proposed in [15]. Due to very high computational complexity, these methods are not suitable for false noise analysis of the large size designs. In [16], an approach based on simple logic implications (SLI) [17] was developed. An SLI expresses a logic relationship between a pair of signals. Initially SLIs are generated for logic gates and then more implications are derived by forward and backward propagation of available SLIs. Paper [18] proposed to use the resolution method, originally used for mechanical theorem proving [19]. The resolution method of [18] was shown to have advantage over [16] as it can model logic correlations between multiple signals and can extract logic correlations from transistor level circuit description without an explicit logic function extraction.

An alternative way to solve the problem of the input vectors search for generating the true critical paths is to construct the so-called timed characteristic function (TCF) or TCF-function [20].

The TCF-function determines a set of input vectors for which the delay exceeds the specified time restriction:

$$TCF(y = v, t_0+) = \{\{\vec{x}\} : \forall t > t_0,\ y(\vec{x}, t) = v\}.$$

The application of the TCF-function in combination with the subsequent analysis of the logical compatibility of the input vector and circuit itself (SAT-analysis) allows finding input actions with the specified delay restrictions. The efficient interaction methods of the TCF-function generator and logical compatibility analyzer [21] by the example of the simple AND, OR, and INV gates are known. Using iterations for different delay restrictions, the true critical path and the corresponding input vector can be found. This paper is the further development of similar idea moving to the following directions:

(i) Firstly, instead of the characteristic function with a one-sided delay restriction, the characteristic interval function is proposed. Such function determines the set of input vectors for which the delay falls into some interval $[a, b]$; in contrast to the TCF approach, this does not require iterations for determining limits of ranges;

(ii) Secondly, the technique for the propagation of characteristic functions along the circuit with the built-in analysis of logical compatibility is proposed. Such technique, in contrast to the TCF approach, does not require the SAT-analysis application; and

(iii) Thirdly, the proposed technique ensures the analysis of logical compatibility of all paths from the specified input switching, including noncritical, and this is of great importance for data preparation during characterization of intellectual property blocks.

The concept of the interval for describing delays, the leading edges of propagation of signals, and input vectors with the Boolean values form the core of the considered approach.

The principles of the interval arithmetic are presented in [22, 23]. The interval simulation methods were substantially developed in the works of the Institute of Theoretical and Applied Mechanics, Novosibirsk, Russia [24, 25]. We note that the up-to-date development of the interval simulation in most cases is intended to solving optimization and stationary problems, described by systems of algebraic equations and inequalities [26, 27]. Less attention is paid to dynamic problems, due to the hard-to-control increase of uncertainty with time.

The similar problem of the increase in uncertainty arises at the logic level in the static timing analysis is going from fixed delays to intervals. In order to overcome this problem, in this work, the following approaches are proposed:

- The concept of the logic-time interval is introduced. It combines the real delay interval at the node of the circuit with the Boolean interval of possible input switching vectors for the delay in the specified interval.

- Specifications of interval characteristic functions (ICFs) are proposed to control the increase in uncertainty of the Boolean intervals.

- Algorithms of the propagation of interval characteristic functions along the circuit based on the BDD technique are proposed [12].

The idea of the incomplete or partial certainty of the Boolean functions was presented in some works oriented at solving logical synthesis and optimization problems [28]. In the present work, the technique of partially specified Boolean functions is used for quick estimation of the compatibility of the input intervals of a particular gate in the process of distributing intervals along the circuit. The final decision on the compatibility of the input intervals is made basing on the exact analysis of their logical compatibility, which, in its turn is based on the proposed technique of interval characteristic functions. Thus, the high algorithm operation rate due to fast estimations based on comparing limits of ranges goes well with the complete analysis of the logical correlations in the circuit due to the ICF technique.

III. Problem Formulation in Terms of Boolean Vector Intervals and Switching Delays Intervals

The combinational circuit can been described in terms of four-digit Boolean algebra:

$$A_4 = (B_4 = B_2 \times B_2, <+>, <*>, <\neg>, <0>, <1>),$$

where B_2 = {0, 1} is the set of the Boolean direct-current states; (< * >, < + >, < ¬ >) are the operations of disjunction, conjunction, and negation in four-digit logic, respectively; < 0 > = (0, 0) is the state of static zero; < 1 > = (1, 1) is the state of static unity; and $B_4 = B_2 \times B_2$ is the Cartesian product of B_2 by itself. In this case, the set of states of circuit nodes is the ordered pairs of the Boolean values $B_4 = \{(x_0, x_1): x_0 \in B_2, x_1 \in B_2\}$ for designations of logic values before and after switching respectively. In order to determine the ordered pairs (x_i, x_j), the following alphabet is used:

$$B_4 = \{L, R, F, H\},$$

where *L* = < 0 > = (0, 0) is the stable state 0 before and after switching (*low*); *R* = (0, 1) is the switch from 0 to 1 (*rise*); *F* = (1, 0) is the switch from 1 to 0 (*fall*); *H* = < 1 > = (1, 1) is the stable state '1' before and after switching (*high*).

Operations of four-digit logic are defined as digit-to-digit operations of two-digit logic for states before and after switching:

$$(a,b) <+> (c,d) = (a \vee c, b \vee d),$$
$$(a,b) <*> (c,d) = (a \,\&\, c, b \,\&\, d),$$
$$<\neg> (a,b) = (\neg a, \neg b).$$

The combinational circuit can be reduced to the required form by extracting the logical functions in the form of the SP graph (SP-DAG) [29] from the description at the transistor level. In the SP graph, the parallel connection corresponds to operation < + >, the series connection corresponds to operation < * >, and the input of the *p*-type transistor or the output of the gate for the pull-down chain from the earth node can correspond to the negation operation < ¬ >.

To determine intervals of possible values of primary inputs, the following vector designations are used:

$$\vec{v} \in [\vec{v}_a, \vec{v}_b] \quad \Leftrightarrow \quad \vec{v}_a \le \vec{v} \le \vec{v}_b,$$

where $\vec{v}_a = | a_1, \dots, a_n |$, $\vec{v}_b = | b_1, \dots, b_n |$ are the lower and upper limits of the ranges, respectively, corresponding in the scalar form for digit-to-digit inequalities for each *n* primary input:

$$a_i \le v_i \le b_i \quad \forall i \in \{1, \dots, n\}.$$

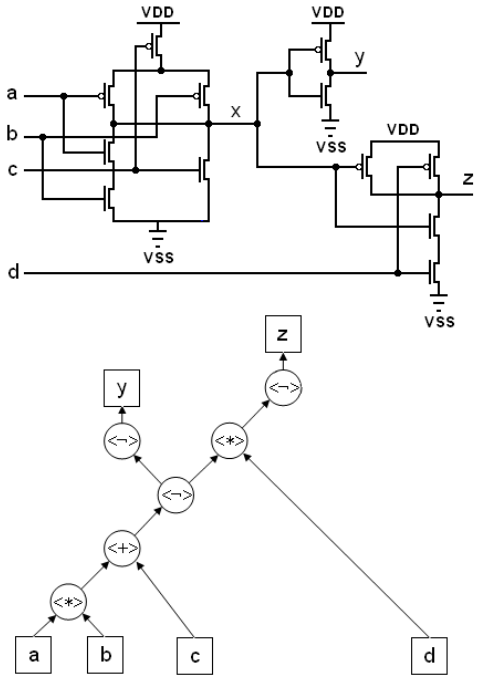


Fig. 1. CMOS circuit and SP-graph

The degree of uncertainty φ(*v*) of the input vector is characterized by the number of unequal values at the limits of the interval:

$$\varphi(v) = \sum_i (a_i \oplus b_i).$$

The traditional logic simulation of the input sequences assumes propagation of logic states along the circuit from the primary inputs to the primary outputs for particular values without intervals of the input vector with the complete control of the circuit's operation logic. In this case, φ = 0, and, vice versa, the traditional static timing analysis method with a search of the critical paths is oriented toward the complete uncertainty in the values of the input vector: φ = *n*. The main idea of this work is the attempt to integrate these two contrasting methods based on the interval approach.

Let the logic-timing interval $L_j(z)$ for the specified circuit node *z* denotes the union of the real interval of possible delay values at the circuit node with the Boolean information on the possible input switching vectors in the following form:

$$L_j(z) = (s_j, [t_{\min}, t_{\max}], [\vec{v}_a, \vec{v}_b]),$$

where $s_j \in B_4$ is the type of the interval in terms of four-digit logic; $[t_{\min}, t_{\max}]$ is the possible delay value interval; and $[\vec{v}_a, \vec{v}_b]$ is the range of possible input vector values, at which the delay is inside the specified interval. In the general case, the set of interval of one type can belong to the specified node.

One of the widespread problems of the logic-timing simulation is to calculate delays at primary outputs of the IP block with the specified switching of one or several primary inputs. Let us assume that the first input is changed from 1 to 0 ($s = F$) and the remaining inputs are in the static state. Then, with the preset period p, if there are no additional restrictions, the following intervals can be determined for primary inputs $x_1, x_2, \ldots, x_n$:

$$\begin{gathered}L_1(x_1) = (F,[0,0],[|F,0,\ldots,0|,|F,1,\ldots,1|]);\\ L_1(x_2) = (L,[0,p],[|F,0,0,\ldots,0|,|F,0,1,\ldots,1|]);\\ L_2(x_2) = (H,[0,p],[|F,1,0,\ldots,0|,|F,1,1,\ldots,1|]);\\ \ldots\\ L_1(x_n) = (L,[0,p],[|F,0,\ldots,0,0|,|F,1,\ldots,1,0|]);\\ L_1(x_n) = (H,[0,p],[|F,0,\ldots,0,1|,|F,1,\ldots,1,1|]).\end{gathered}$$

The interval simulation problem will extend the intervals from the primary inputs through intermediate nodes to outputs of the circuit.

IV. Intrval Propagation across the Circuit

Since the circuit is based on the SP graph with the use of four-digit logic operations, it is necessary to form generation expressions of the new intervals at outputs $y = x_1 {<}+{>} x_2$; $z = x_1 {<}*{>} x_2$ for each pair of intervals at the inputs. Let

$$\begin{gathered}L_1(x_1) = (s_1,[l_1,r_1],[\vec{a}_1,\vec{b}_1]),\\ L_2(x_2) = (s_2,[l_2,r_2],[\vec{a}_2,\vec{b}_2]),\end{gathered}$$

then the type of output interval is formed in accordance with the four-digit logic expressions:

$$\begin{gathered}s(y) = s(x_1) < + > s(x_2);\\ s(z) = s(x_1) < * > s(x_2).\end{gathered}$$

It is evident that the logic compatibility of the input intervals is possible only in the area of their intersection; therefore, the Boolean intervals for y and z are formed as

$$[\vec{a}_1,\vec{b}_1] \cap [\vec{a}_2,\vec{b}_2] = [\vec{a}_1 \vee \vec{a}_2, \vec{b}_1 \,\&\, \vec{b}_2].$$

The intersection of the Boolean intervals can prove to be empty if the collision condition pointing to the detection of the false path (in this case, the new interval is not formed) is fulfilled:

$$(\vec{a}_1 \,\&\, \vec{a}_2 \,\&\, \neg(\vec{b}_1 \vee \vec{b}_2)).$$

The negation operation corresponds to the gate output in the SP graph; therefore, for each input interval $L_i(x) = (s_x,[l_x,r_x],[\vec{a}_x,\vec{b}_x])$, the output interval of the opposite type with the same Boolean restrictions, but with biased limits of the delay interval is formed at the gate output $y = {<}\neg{>} x$ with a delay in the interval $[t_{\min}, t_{\max}]$:

$$L_i(y) = ({<}\neg{>} s_x,[l_x + t_{\min}, r_x + t_{\max}],[\vec{a}_x,\vec{b}_x]).$$

Despite screening the false intervals, the intervals at the outputs of binary operations are formed for each pair of input intervals, leading to their exponential growth. In order to reduce the growth of the interval number, it is possible to restrict the maximum number of intervals of each type by limit value $I_{\max}$, and, in order to fulfill this restriction, the union of close intervals can be used (by analogy with method [27] for delay interval limits):

$$[\vec{a}_1,\vec{b}_1] \cup [\vec{a}_2,\vec{b}_2] \subseteq [\vec{a}_1 \,\&\, \vec{a}_2, \vec{b}_1 \vee \vec{b}_2].$$

Using the union operation and handling with presetting the limit value $L_{\max}$, it is possible to obtain different results with a different degree of the evaluation of the circuit's operation logic. In particular, when $L_{\max} = 1$, two extreme delay values $[t_{\min}, t_{\max}]$ are obtained at the output virtually without evaluating the circuit's operation logic, corresponding to the results of the static timing analysis. In contrast, when $L_{\max} = \infty$, the union of intervals is not performed, and the results correspond to the complete simulation of all possible input stimulus.

The important problem, related to the operation of the union, consists in the fact that an ambiguity can arise, namely, switching vectors, which do not actually correspond to the interval specification (type and delays), and fall into the integrated interval. More detailed evaluation of the circuit's operation logic is necessary to solve this problem.

V. The Technique of Interval Characteristic Functions

In order to control the growth of uncertainty of the Boolean intervals, specifications of interval characteristic functions (ICF) and propagation algorithms of the ICFs along the circuit, based on the BDD technique, are proposed.

The technique of partially specified the Boolean functions with the use of vector boundaries is efficient for quick estimation of the compatibility of the input intervals of a particular gate in the process of interval propagations along the circuit. This technique ensures the completeness of the logic compatibility analysis until the intersection operations are used and the union operations are not performed. The concept of the ICF is introduced to preserve completeness of the information on the input vectors of a particular interval.

For logic-timing interval $L_j(z) = (s,[t_{\min},t_{\max}],[\vec{v}_a,\vec{v}_b])$, the interval characteristic function ICF(L_j) is defined as the Boolean function, whose arguments are the Boolean values of the variables (of primary inputs after switching), and the function value is equal to 1, including the case when the input vector ensures switching (or state), corresponding to specifications s, $[t_{\min}, t_{\max}]$.

For the intersection and union of the input intervals, the characteristic function of the new interval can be composed using the BDD technique [12], as the conjunction and disjunction of characteristic functions of the input intervals:

$$\begin{gathered}ICF([\vec{a}_1,\vec{b}_1] < * > [\vec{a}_2,\vec{b}_2]) = ICF([\vec{a}_1,\vec{b}_1] < + > [\vec{a}_2,\vec{b}_2]) =\\ = ICF([\vec{a}_1,\vec{b}_1] \cap [\vec{a}_2,\vec{b}_2]) = ICF([\vec{a}_1,\vec{b}_1]) \,\&\, ICF([\vec{a}_2,\vec{b}_2]),\\ ICF([\vec{a}_1,\vec{b}_1] \cup [\vec{a}_2,\vec{b}_2]) = ICF([\vec{a}_1,\vec{b}_1]) \vee ICF([\vec{a}_2,\vec{b}_2]).\end{gathered}$$

The characteristic function which is the result of the negation operation is identical to the characteristic function of the input (argument). The BDD technique is efficient for this approach to eliminate false paths, since, in this case, the collision condition is determined as identical to 0:

$$ICF([\vec{a}_1,\vec{b}_1]\cap[\vec{a}_2,\vec{b}_2])\equiv 0.$$

VI. Proposed Delay Model to Account for the Inputs Simultaneous Switching

Normally the static timing analysis considers only one input gate switching with fixed conditions at the other inputs. But the delay of element is substantially reduced for the simultaneous controlling input switching due to activation of multiple paths conduction current/charge value at several inputs [30]. Therefore, to improve the calculation accuracy of the interval delay or minimum boundaries, it is necessary to consider the library element simultaneous input switching delay. However, accurate analysis of inputs simultaneous switching is required to move from the two-dimensional NLDM (Non-Linear Delay Model) tables to the four-five-dimensional account for all fronts of switching inputs [1, 30]. To reduce the dimensions of the problem, we propose to apply cubic delay approximation:

$$\Delta_D = c_1x^3 + c_2y^3 + c_3x^2y + c_4xy^2 + c_5x^2 + \\ + c_6y^2 + c_7xy + c_8x + c_9y + c_{10}, \qquad (1)$$

where $x = S_x$, $y = S_y$ – the duration of switching inputs, c_i – approximation coefficients. The method of least squares is used to find the coefficients.

The simulation results without the simultaneous inputs switching are compared with the simulation results based on simultaneous switching to improve the accuracy of minimum delay estimates. The difference between the two dependencies can be found from the formula:

$$\Delta_D = \min(D_1, D_2) - D_{\min},$$

where D_1, D_2 is delay of input switching signals x and y respectively, Δ_D - correctional difference between delay element without and with considering simultaneous inputs switching.

Application of correctional difference cubic approximation Δ_D allows increasing the accuracy of the minimum delay calculation, which can be found by the formula:

$$D_{\min} = \min(D_1, D_2) - \Delta_D.$$

Minimum of delay element is achieved at simultaneous control switching at the inputs of logic element. For example for nand2 controlling switching it is switching from 1 to 0. In the case of a complex element the search of the minimum delay requires to take account its series-parallel structure.

Consider finding minimum delay on the example of the element nand2, at simultaneously opening of the parallel-connected transistors (pull-up chain) the rise delay can been found by the formula: $D_{\min} = \min(D_a, D_b) - \Delta_D$. In other cases, a minimum fall delay can been found by the classic formula: $D_{\min} = \min(D_a, D_b)$ (fig. 2).

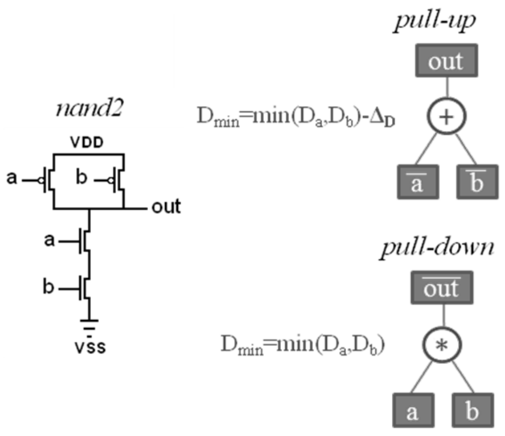


Fig. 2. Fall and rise minimal delay recalculation

To determine the minimum delay it is necessary to analyze a gate series-parallel structure. For example, for element aoi21 the minimum delay can be found by means of the SP-DAG graph navigation from bottom to top for pull-up and pull-down networks independently. For pull-up network (Fig. 3), where inputs a and b are switching simultaneously: $D_{\min}(a,b) = \min(D_a,D_b) - \Delta_D$, then the resulting delay is compared with the input switching delay c: $D_{\min} = \min(D_{\min}(a,b), D_c)$.

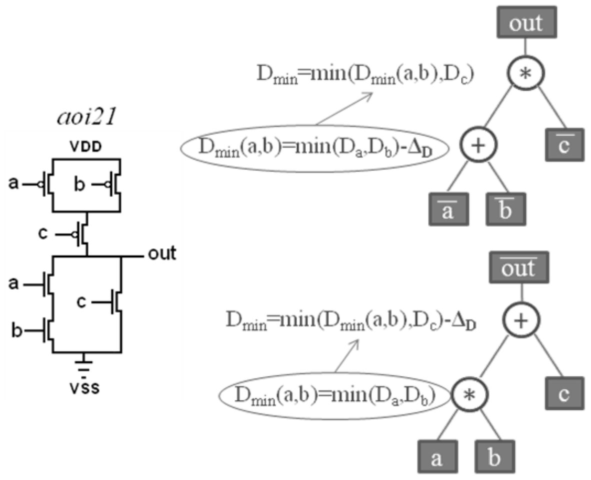

aoi21
VDD
a
b
c
out
a
c
b
VSS
out
$D_{\min}=\min(D_{\min}(a,b),D_c)$
*
$D_{\min}(a,b)=\min(D_a,D_b)-\Delta_D$
+
$\bar{c}$
$\bar{a}$
$\bar{b}$
$\overline{out}$
$D_{\min}=\min(D_{\min}(a,b),D_c)-\Delta_D$
+
$D_{\min}(a,b)=\min(D_a,D_b)$
*
c
a
b

Fig. 3. SP-DAG for pull-up and pull-down of aoi21

VII. Results of Numerical Experiments

The proposed algorithms are implemented as the part of the logic-timing analysis tool. Experiments were done for ISCAS-85 circuits and for some industrial circuits. The efficiency of the algorithms was estimated in comparison to the obtained parameters of the static timing analysis, reduction in false path number, and reduction in delay interval limits - the average and maximum reduction of the maximum delay and the average and maximum increase in the minimum delay.

Based on the numerical experiments, it was shown that the proposed method reduces the number of false paths up to 35%; while the logic evaluation is intended to reduce timing intervals on average by 7% for the maximum limit and by 5%

for the minimum limit. In this case, in some situations, the decrease of the maximum delay reaches 90% and the minimum delay is more than double. The algorithm's operation time on the Intel Core Quad CPU Q8300 2.5 GHz for the above listed circuits was less than 1 min per circuit, allowing usage of this approach in the optimization procedures.

Based on the numerical experiments, the proposed approach grants more authentic results, compared to the normal static timing analysis.

VIII. Conclusion

In this paper we propose the method for IP blocks performance analysis to improve the accuracy of analysis of minimum and maximum delay at the logical and system level. The developed logic-timing simulation method for CMOS circuits based on the interval estimation ensures integration of two contrasting approaches to solve the problem of the performance analysis (analysis of critical paths and simulation of input stimulus). Thus, the analysis of logically compatible paths, combining the speed of the algorithm for the critical path analysis and the accuracy of the test sequence simulation, is reached. Compared to the known analysis methods of logically consistent critical paths (such as TCF and Sensitization), the proposed approach ensures the analysis of the logical compatibility of all paths from the preset input switching, does not require iterations for determining interval limits, and allows to use exact delay models. The algorithm of propagation characteristic functions along circuit with the analysis of the intervals logical compatibility is proposed. This method allows to reduce a number of false paths and to improve the accuracy of delay estimation in comparison with normal static timing analysis. To improve the interval estimation accuracy for the minimum delay border, the correctional difference cubic approximation technique was proposed to take into consideration the simultaneous input switching.

References

[1] S.V. Gavrilov, O.N. Gudkova, A.L. Stempkovskiy. The Analysis of the Performance of Nanometer IP-blocks Based on Interval Simulation // Russian Microelectronics, Vol.42, N7, 2013, P. 396–402.

[2] Sergey Gavrilov, Galina Ivanova, Pavel Volobuev, Aram Manukyan. Methods of logical synthesis for library elements and blocks with regular layout structure // 2015 IEEE 35th International Conference on Electronics and Nanotechnology (ELNANO-2015), 2015, P. 138-141.

[3] Gavrilov, S., Glebov, A., Soloviev, R., et al., Delay noise pessimism reduction by logic correlations, Proc. of ICCAD, 2004, pp. 160–167.

[4] Gavrilov, S.V., Glebov, A.L., and Stempkovskiy, A.L., Methods for increasing efficiency of VLSI timing analysis, Informational Technologies, 2006, no. 12, pp. 2–12.

[5] Gavrilov, S.V., Glebov, A.L., and Stempkovskiy, A.L., Metody logicheskogo i logiko-vremennogo analiza tsifrovykh KMOP SBIS (Methods of logic and logic-timing analysis of digital CMOS VLSI circuits), Moscow: Nauka, 2007.

[6] Silva, J.P.M. and Sakallah, K.A., Efficient and robust test generation-based timing analysis, Proc. ISCAS, 1994, pp. 303–306.

[7] Gavrilov, S., Glebov, A., Sundareswaran, S., et al., Accurate input slew and input capacitance variations for statistical timing analysis, Proc. of Austin Conf. on Integrated Systems & Circuits, Austin, 2006.

[8] S. V. Gavrilov, O. N. Gudkova and Yu. B. Egorov. Methods of accelerated characterization of VLSI cell libraries with prescribed accuracy control // Russian Microelectronics, Vol.40, N7, 2011, P.476-482.

[9] Sergey Gavrilov, Olga Gudkova, Roman Soloviev. Timing Analysis for Complex Digital Gates and Circuits Accounting for Transistor Degradation // Proceedings of SEUA. Series “Information technologies, Electronics, Radio engineering”. Issue 16, N1, 2013, P.84-93.

[10] Coudert, O., An efficient algorithm to verify generalized false path, In Proc. of DAC, 2010, pp. 188–193.

[11] Bahar, R.I., Cho, H., Hachtel, G.D., et al., Timing analysis of combinational circuits using ADD's, Proc. of IEEE European Design Test Conference, 1994, pp. 625–629.

[12] Bryant, R.E., Graph-based algorithms for Boolean function manipulation, IEEE Trans. on Computers, 1986, pp. 677–691.

[13] Shepard K.L. “Design methodologies for noise in digital integrated circuits”, Proc., DAC, 1998, pp. 94-99.

[14] Gavrilov, S.V., Glebov, A.L., and Stempkovskiy, A.L., Methods for increasing efficiency of VLSI timing analysis, Informational Technologies, 2006, no. 12, pp. 2–12.

[15] Gavrilov, S., Glebov, A., Soloviev, R., et al., Delay noise pessimism reduction by logic correlations, Proc. of ICCAD, 2004, pp. 160–167.

[16] A. Glebov, S. Gavrilov, D. Blaauw, S. Sirichotiyakul, C. Oh, V. Zolotov, “False noise analysis using logic implications”, ICCAD 2001, pp. 515-521.

[17] F.M.Brown. “Boolean reasoning”, Kluwer Academic Publishers, 1990.

[18] A. Glebov, S. Gavrilov, D. Blaauw, V. Zolotov, R. Panda, C. Oh “False-noise analysis using resolution method” ISQED 2002, pp. 437-442.

[19] J.A.Robinson A. Machine-Oriented Logic Based on the Resolution Principle, J. of the ACM, 12(1): 23-41, 1965.

[20] Yu-Min Kuo and Yue-Lung, Efficient Boolean characteristic function for fast timed ATPG, ICCAD'06, 2006, pp. 96–99.

[21] Young, R.C., Algebra of many-valued quantities, Mathematische Annalen., 1931, Bd. 104, pp. 260–290.

[22] Warmus, M., Calculus of approximations, Bull. Acad. Polon. Sci., 1956, Cl. III, vol. IV, no. 5, pp. 253–259.

[23] Sunaga, T., Theory of an interval algebra and its application to numerical analysis, RAAG Memoirs., 1958, vol. 2, Misc. II, pp. 547–564.

[24] Shokin, Yu.I., Interval'nyi analiz (Interval analysis), Novosibirsk: Nauka, 1981.

[25] Kalmykov, S.A., Shokin, Yu.I., and Yuldashev, Z.Kh., Metody interval'nogo analiza (Methods of the interval analysis), Novosibirsk: Nauka, 1986.

[26] Hansen, E. and Walster, G.W., Global optimization using interval analysis, N.Y.: Marcel Dekker, 2004.

[27] Bobba, S., and Hajj, I.N., Estimation of maximum current envelope for power bus analysis and design, Int. Symp. on Phys. Des., 1998, pp. 141–146.

[28] Sakallah, K.A., Functional abstraction and partial specification of Boolean functions, The University of Michigan, 1995.

[29] Bryant, R.E., Boolean analysis of MOS circuits, IEEE Transactions on Computer-Aided Design of Integrated Circuits, 1987, pp. 634–649.

[30] L.-C. Chen, S. K. Gupta, and M. A. Breuer, "A new gate delay model for simultaneous switching and its applications," in Proc. Design Automation Conference, 2001, pp. 289-294.

Bridging the Gap Between Probabilistic Safety Analysis and Fault Injection in Virtual Prototypes

Moomen Chaari*†, Bogdan-Andrei Tabacaru*†, Wolfgang Ecker*†, Cristiano Novello*, and Thomas Kruse*
*Infineon Technologies AG - 85579 Neubiberg, Germany
†Technische Universität München
Firstname.Lastname@infineon.com

***Abstract*—Safety evaluation has become a non-negotiable task in many fields of system design, particularly in the automotive industry. Nowadays, systems are explored with respect to their behavior in failure cases throughout the design and manufacturing cycle. On the early stages, a probabilistic analysis is performed, based on the conceptual description of the system. Later on, when the implementation details are available, a fault-effect-simulation is carried out through fault injection campaigns on an executable system model. Traditionally, the probabilistic safety analysis on the one hand and the fault-effect-simulation on the other hand are accomplished by different teams and in compliance with different flows and methodologies.**
In this paper, we introduce a model-based approach to connect these two aspects of system safety evaluation. Our solution enables a formalized and systematic data exchange between both contexts leading to considerable effort reduction reaching up to 60%. It also allows an automated stimulation of fault injection campaigns using safety analysis artefacts and outcomes, and offers a feedback mechanism from the simulation to the probabilistic analysis.

Keywords*—*safety analysis, model-driven development, fault simulation, fault injection, metamodeling.

I. Introduction

Integrated circuits are used nowadays in a wide range of safety-critical embedded systems whose operation may affect human life. Therefore, safety assessment represents a key step in the design and manufacturing cycle for the semiconductor industry. Through this assessment, the risk of an erroneous behavior having a critical impact on system safety is minimized. To achieve this, designers and manufacturers follow the safety evaluation guidelines given by the relevant standards to their respective domains. For example, in the automotive context, these guidelines are addressed by the ISO 26262 standard for functional safety of road vehicles [3], which has been built upon the IEC 61508 standard [2].

To certify an electrical/electronic (E/E) system as functionally safe in compliance with the ISO 26262 standard, an extensive procedure is required. It starts by predicting potential risks, whose causes and effects are subsequently identified. In order to mitigate the failure effects, appropriate countermeasures are deployed. And finally, evaluation metrics are computed to provide evidence about the system safety level. This *probabilistic* safety analysis process relies on manual inspection and data gathering, expert judgement, and spreadsheet based calculations. It is performed using different methods such as Failure Modes and Diagnostic Analysis (FMEDA), Fault Tree Analysis (FTA), and Dependent Failure Analysis (DFA). When the targeted safety level of the considered system is particularly high, the standard prescribes the additional application of fault injection techniques. For this, faults are deliberately inserted into an executable system model which is afterwards monitored to determine its behavior in response to the introduced faults. Running the modified system model on an appropriate simulation platform enables the perception of the actual failure effects, the detection of the faulty scenarios leading to critical system failures, and the qualification of the safety/diagnostic mechanisms which are already integrated in the system model. Thus, fault injection is considered as a *model-based*, *quantitative* safety assessment process as it relies on the system model and delivers measured values for failure rates and coverage metrics.

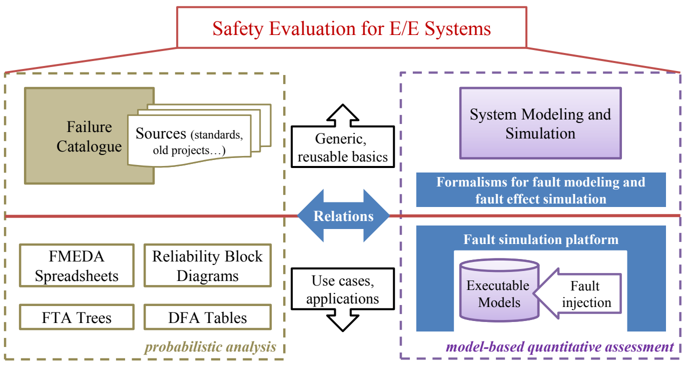

Safety Evaluation for E/E Systems
Failure Catalogue
Sources (standards, old projects…)
Generic, reusable basics
System Modeling and Simulation
Formalisms for fault modeling and fault effect simulation
Relations
FMEDA Spreadsheets
Reliability Block Diagrams
FTA Trees
DFA Tables
Use cases, applications
Fault simulation platform
Executable Models
Fault injection
probabilistic analysis
model-based quantitative assessment

Fig. 1: Different Perspectives of Safety Evaluation

Figure 1 gives an overview of the two safety evaluation perspectives that have been mentioned above: the probabilistic analysis and the model-based quantitative assessment. For each perspective, we differentiate between the reusable generic basics and the application-dependent use cases.

II. Proposed Approach

In our work, we investigate the relations between the two safety evaluation perspectives shown in Figure 2. To bridge the gap between the probabilistic analysis (FMEDA, FTA, DFA,...) and the quantitative assessment (fault-injection on executable system models), we introduce a model-based alternative to the traditional safety analysis procedure. The application of model-driven development techniques, particularly metamodeling and code-generation, allows a structural formalization of safety analysis methods and artefacts, enables a systematic extraction of relevant data for the analysis, and offers speed-up and quality improvement by reducing cumbersome and error-prone

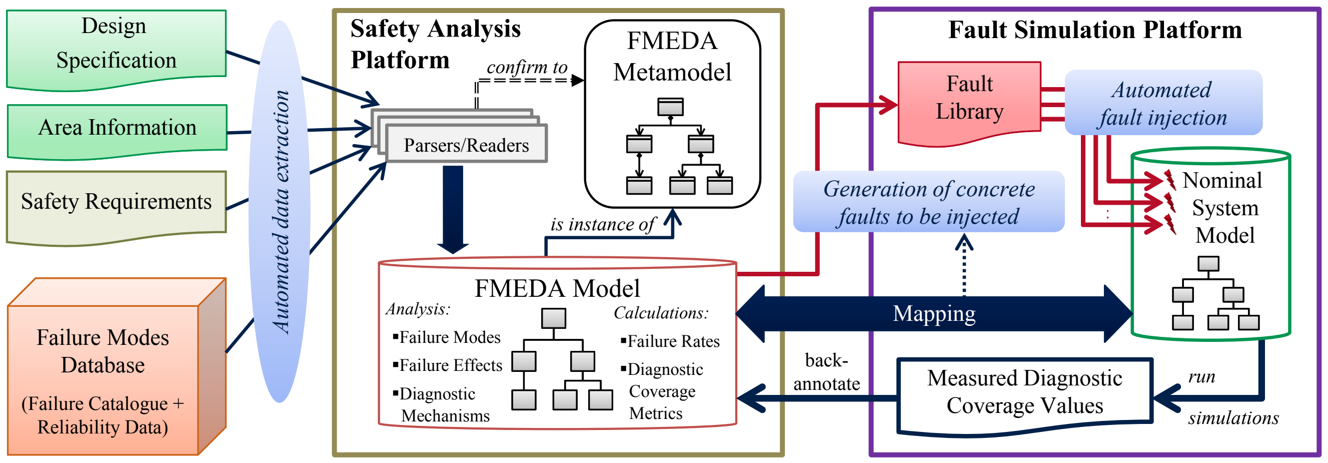


Fig. 2: Approach Overview: Model-Based FMEDA and Link to Fault Injection and Simulation

manual tasks. In [1], we describe a model-based automation approach for FMEDA and present the corresponding FMEDA metamodel which is constructed as a formal description of FMEDA elements and relationships between them. A similar formalization process is performed for FTA and DFA. In this paper, we focus on the link between FMEDA and fault simulation. The according approach overview is illustrated in Figure 2. To substitute the traditionally used FMEDA spreadsheet, an FMEDA data model is built as an instance of the FMEDA metamodel. The inputs which are conventionally taken into account by the safety analyst remain unchanged: the design specification, the area information, the application-dependent safety requirements, and the failure modes database which combines the failure catalogue contents and the reliability data (see Figure 2). The traditional manual data input is replaced by an automated data extraction using a set of parsers and readers which confirm syntactically to the FMEDA metamodel. This data model which covers the analysis artefacts (failure modes, failure effects, and diagnostic mechanisms) and the according calculations (failure rates and diagnostic coverage metrics) is the key to link the FMEDA to the simulation context. Through model-to-model mapping, specific elements of the FMEDA data model are linked to the appropriate elements of the nominal system model to be simulated. For example, failure modes are mapped to injection points, failure effects to observation points, and diagnostic mechanisms to diagnostic points. It should be noted that the so-called injection/observation/diagnostic points are ports, signals, variables, etc., and are determined by matching algorithms applied on the FMEDA model and the executable system model. The outcome of the mapping is used to generate a fault library listing the concrete faults to be injected in the system model. The diagnostic coverage values which are measured throughout the corresponding simulation runs are back-annotated into the FMEDA model enabling a potentially required refinement of safety measures when the target safety level is not reached.

III. Application Example and Results

To evaluate the applicability and the benefits of the approach described above, a CPU model addressing the MIPS (Microprocessor without Interlocked Pipeline Stages) architecture and offering a reduced instruction set (about 60 instructions) is considered. The model-based automation support for FMEDA, relying on (i) the formalized FMEDA metamodel, (ii) the confirming tools and utilities, and (iii) the reusable failure modes database, offers an effort reduction reaching up to 60% in comparison to the manual analysis. A SystemC virtual prototype of the considered CPU is used for the fault injection and simulation. The successful mapping of the FMEDA data model to the SystemC model makes it possible to generate the fault library and to obtain a traceability between the analysed failure modes and the correspondingly injected faults. In addition to that, time savings are achieved by efficient fault injection and simulation in SystemC and the back-annotation of the measured diagnostic coverage values allows an early detection of deficiencies in the planned safety measures.

IV. Conclusion

In this work, we address the problematic divergence between probabilistic safety analysis and model-based fault injection. As both evaluation aspects are considered for the certification of systems deployed in safety-critical domains, a systematic data exchange between them is required. Through model-driven development techniques, a formalization of safety analysis methods is achieved and a link to fault injection and simulation is established. A considerable speed-up of the safety evaluation cycle is realized and the costs for merging, aligning, and reporting evaluation artefacts coming from different contexts are significantly reduced.

References

[1] M. Chaari, W. Ecker, C. Novello, B.-A. Tabacaru, and T. Kruse. A Model-Based and Simulation-Assisted FMEDA Approach for Safety-Relevant E/E Systems. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6. ACM, 2015.

[2] International Electrotechnical Commission and others. Functional safety of electrical/electronic/programmable electronic safety related systems. *IEC 61508*, 2000.

[3] ISO, CD. 26262, Road vehicles–Functional safety. *International Standard ISO/FDIS*, 26262, 2011.

Using Virtual Platform for Reliability and Robustness Analysis of HW/SW Embedded Systems

Reda Nouacer
CEA, LIST
Software Reliability and Security Laboratory
P.C. 174, Gif-sur-Yvette, 91191, France
reda.nouacer@cea.fr

Manel Djemal
LAMIH
University of Valenciennes
59300 Valenciennes, France
manel.djemal@univ-valenciennes.fr

Smail Niar
LAMIH
University of Valenciennes
59300 Valenciennes, France
smail.niar@univ-valenciennes.fr

I. Motivation

It is largely recognized that the architectures of embedded systems in transportations tools are becoming more and more complex. Due to the constant advances in microelectronic (Fig. 1), embedded system engineers are now able to integrate more system functions on powerful System-on-Chips (SoCs). The transportation systems industries also benefit from advances in microelectronics and embedded system engineers in these areas are now able to integrate all the vehicle electronic functions on fewer ECUs. These systems are becoming complex, distributed and include an increasing number (from 10 to 100) of Electronic Control Units (ECU). This trend, however, raises the issue of the robustness and reliability of these systems, due to the increase in the error ratio with the level of integration, the clock frequency and the functioning conditions, such as temperature, fields magnetic, etc.

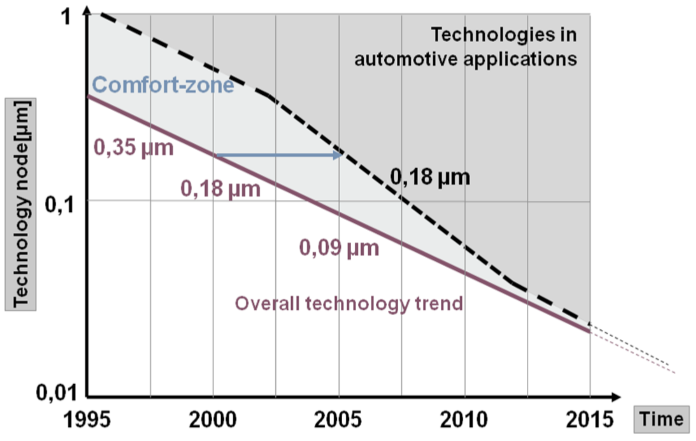


Fig. 1: CMOS technology trend used for automotive applications [1]

Robustness and fault tolerance are critical in the automotive domain for several reasons:

1) Embedded systems in transport systems are exposed to magnetic fields and radiation that are much important than those found in conventional mobile systems such as in smartphones.
2) Embedded applications in transportation systems ensure first-order security features and can cause loss of human lives.

Taking into account reliability in the design of next embedded software and hardware in transportation systems will be crucial. The ISO 26262 standard for instance has introduced stricter safety requirements in the automotive field such as DO254 standards and DO178C do in avionics.

II. Yet another fault injection based approach ?

Existing tools for testing embedded systems in critical systems can be classified as either for functional testing or robustness testing. Functional testing, is a black box testing in which systems are tested by feeding their input and examining their output. In other words, the system is tested to ensure that it conforms to all its functional requirements. Robustness testing consists in observing the behavior of a system under exceptional execution conditions.

Taking into account hardware failures by the embedded software is a very complex task and needs more attention. Most of the earlier works were merely focused either on software reliability with no consideration of the hardware part or vice versa. In the existing approaches, during the HW/SW integration phase, functional tests are performed on the entire system as a black box, but without considering hardware failures (Fig. 2). This poses a serious problem for manufacturers. Thousands of hours of functioning on the real system are needed to meet the safety standard requirements and operating security.

Evaluating the reliability and the robustness of a processing architecture is a complex task. Among the other possible approaches, we cite analytical modeling whose applicability is very difficult due to the fact that the occurring faults phenomena is relatively random. Fault injection [2], also known as fault insertion testing, consists in perturbing the target system with permanent, intermittent faults or transient faults, and monitoring the system to determine its behavior in response to a fault.

Unlike other proposed approaches, where the fault injection is random, we made the choice of determinism. On one hand, the calculation of the probability of occurrence of the hardware fault depends on hardware physical characteristics and aging of the under-study embedded system. On the other

This work was financially supported by Bpifrance AAP FUI16 project EQUITAS and the General Counsel of Essonne - France

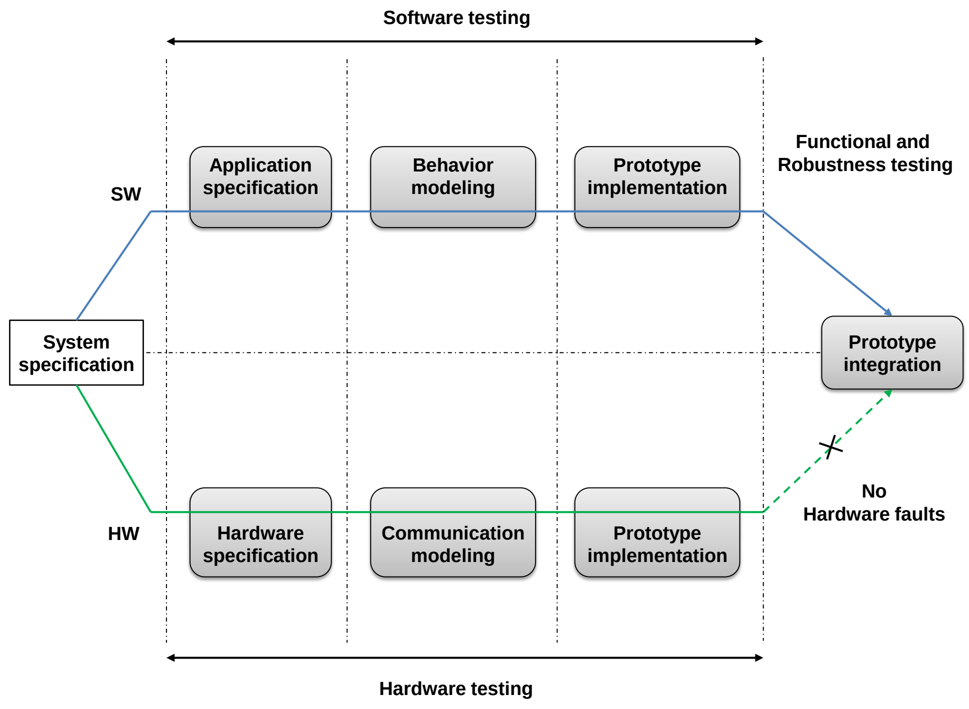


Fig. 2: HW/SW system design and testing flow

hand, the injection of the fault is driven by the test scenario and hence enables choosing the perimeter of the study. We have also chosen to implement our model in a UNISIM-VP [3][4] simulation environment (BSD license) which is easily expandable thanks to a component-based software architecture.

III. Related works

A large number of works has been made to develop techniques for injecting faults into a system prototype or a model. There are four main categories: hardware-based, software-based, simulation-based, and emulation-based fault injections. Hardware-based injection uses additional hardware to introduce faults into the target hardware. Depending on the faults and their locations, there are two hardware-based fault injection methods [7]:

- Hardware fault injection with contact: The fault is injected using a direct physical contact with the target system by introducing voltage or current pulses into the circuit, or modifying the value of the circuit pins.
- Hardware fault injection without contact: There is no direct physical contact with the target system. For example, disturbing the hardware with parameters of the environment (heavy ion radiation, electromagnetic interferences, etc.).

Hardware-based fault injection technique present several benefits, such as it can access locations that are hard to be accessed by other means, it is better suited for the low level fault models, and experiments are fast. However, this method can introduce high risk of damage for the injected system, and presents limited observability and controllability.

Software-based fault injection [7] is used to inject faults into the operation of software. Several kinds of faults may be injected. For example, register and memory errors, dropped or replicated network packets and erroneous flags. Software fault injection techniques are attractive because they dont require expensive hardware. Although the flexibility of this approach, it follows some drawbacks: it cannot inject faults into locations that are inaccessible to software, and it requires a modification of the source code that may disturb the workload running on the target system.

Simulation-based fault injection [8] consists in injecting the faults in high level models (most often VHDL models). It involves the construction of a simulation model of the system, including a detailed simulation model of the processor in use. But this method demands a large development effort.

Finally, in the emulation-based fault injection, the circuit to analyze is implemented on the FPGA. The synthesized circuit is connected to a computer which role is to control the fault injection triggers and display the results. This technique has same drawback as hardware-based fault injection technique.

In our case, we use the simulation-based fault injection approach by injecting faults in a given architecture at the simulation level in order to measure the error ratio, and to evaluate the target system robustness.

IV. Reliability and Robustness Analysis

In this paper, we will describe a methodological approach to study the reliability of a complete system, i.e. with its HW and SW parts, by merging functional testing on virtual platform and hardware fault injection. The use of virtual platforms aims to offer efficient instrumentation possibilities that are not possible on a real system. In addition, it allows observing the behavior of the system in presence of faults by the injection of hardware faults at different steps of the HW design. The extension of UNISIM-VP focuses on the modeling of hardware faults (transient and permanent) in different simulated units (processors, SRAM, bus, I/O interface, etc.) by implementing new services and interfaces for hardware fault injection (Memory Fault Injection, CAN Fault Injection, etc.). The test cases are generated by MaTeLo tool [5]. In MaTelo, these test cases represent a large number of real-life situations: their execution allows a user to obtain an experimental measurement of the systems operational reliability.

The proposed approach (Fig. 3), described hereafter, takes as inputs the application, the target hardware and a set of system requirements. The fault injection model take into account the target hardware characteristics, using the IEC61709:2011 standard [6] as a reference. We have defined the following services and the corresponding interfaces:

1) Fault quantification: We define a physical model of the fault by studying the architectural features of the hardware target and a set of parameters/phenomena associated with it (e.g. the fine engraving, semiconductor, protection of the integrated circuit, age, temperature, frequency, etc.). This model calculates the probability of occurrence of a fault.

2) Fault injection: Once the fault model is defined, the test scenarios base is instrumented in order to fix the relevant injection points for the current use case. These injection points will serve as triggers during the scenario execution to inject hardware faults.

3) Execution traces comparator: Three time stamped traces are generated as a result of the simulation: program execution trace, monitored data trace, and injected faults trace. Program trace contains a list of executed instruction. Data trace contains a list of monitored data (program symbol) with values and the occurred operations (read/write) along simulation. Fault trace lists the injected faults with the triggering probability. All these traces are needed as a data support during reliability quantification.

4) Reliability quantifier: Having the possibility to inject faults at the different HW components of the embedded system, we need to detect and analyze their impacts. This corresponds to the degree of propagation of the injected faults and the probability for the fault to generates an error of a failure. Once an error or a failure is detected, the embedded system behavior is then analyzed. The purpose here is to identify the robustness level by analyzing the effects of the injected fault and its probability of occurrence in order to take corrective decisions, or to compute error/failure ratio (such as the MTTF) when a system failure happens as a consequence of the injected fault.

V. Fault quantification

Fault modeling (i.e quantification) is a crucial step towards proposing fault injection platforms. Hence, building an accurate fault model is an imperative to represent correctly how faults occur in reality comparing to random methods. The proposed fault model is mainly based on FMEA (Failure Modes and Effects Analysis) technique. This technique provides a combination between failures and their impacts on the system. Quantitatively, the reliability of a device is expressed by its reliability function R(t) which is the probability that the device will operate correctly from time *zero* to time *t*. An alternative way of expressing device reliability is by its failure rate λ(t), which represents the rate of failures per unit of time.

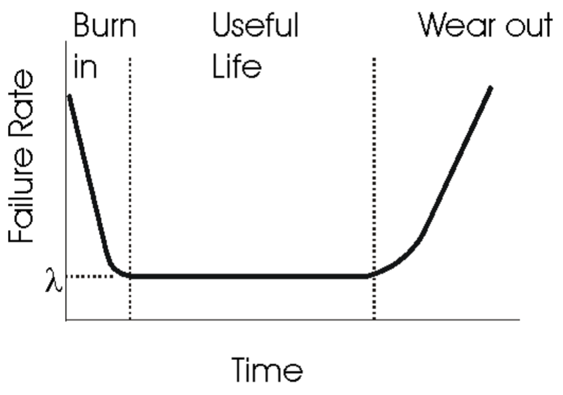


Fig. 4: Component life-time reliability

The bathtub curve shown in (Fig. 4) gives the evolution in time of the failure rate. It is a manufacturer's responsability to ensure that product in the 'infant mortality period' does not get to the customer. This leaves a product with a useful life period during which failures occur randomly i.e λ is constant, and finally a wear out period, usually beyond the product useful life, where λ is increasing.

During the 'useful life period' assuming a constant failure rate, MTBF (Mean Time Between Failures) is the inverse of the failure rate and we can use the terms interchangeably, i.e.

$$\lambda = 1/MTBF \qquad (1)$$

where MTBF is the average time between failures of a system (for reparable systems).

When the failure rate λ is constant, the reliability is statstically expressed by:

$$R(t) = 1 - P(failuretime \leq t) = exp(-\lambda t) \qquad (2)$$

where *P(t)* represents the probability that the system will not conform to its specification throughout a duration *t*.

To study HW/SW system reliability and robustness, we need to generate a series of values λi. In the proposed framework we intend to use the IEC61709:2011 standard [9] for hardware fault quantification.

VI. Fault injection

Fault injection (Fig. 5) is driven by the test scenario and fault configuration.

Once the fault model is defined, the test scenarios database is instrumented in order to fix the relevant injection points for the current use case. These injection points will serve as triggers during the scenario execution to inject hardware faults.

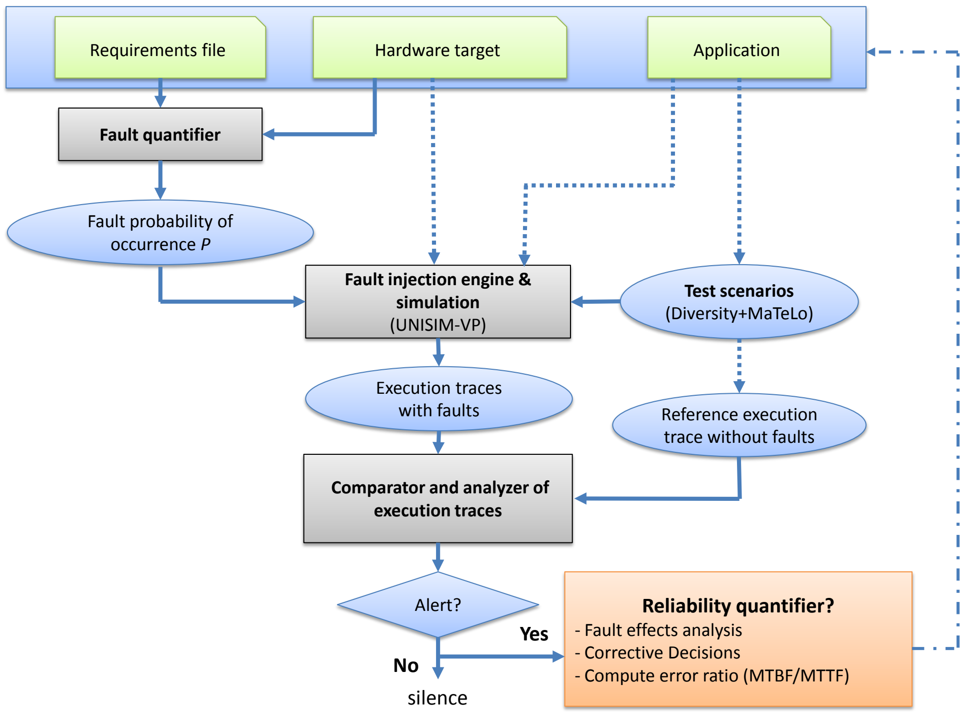

Requirements file
Hardware target
Application
Fault quantifier
Fault probability of occurrence P
Fault injection engine & simulation (UNISIM-VP)
Test scenarios (Diversity+MaTeLo)
Execution traces with faults
Reference execution trace without faults
Comparator and analyzer of execution traces
Alert?
Yes
No
silence
Reliability quantifier?
- Fault effects analysis
- Corrective Decisions
- Compute error ratio (MTBF/MTTF)

Fig. 3: Hardware fault injection in design and test flow

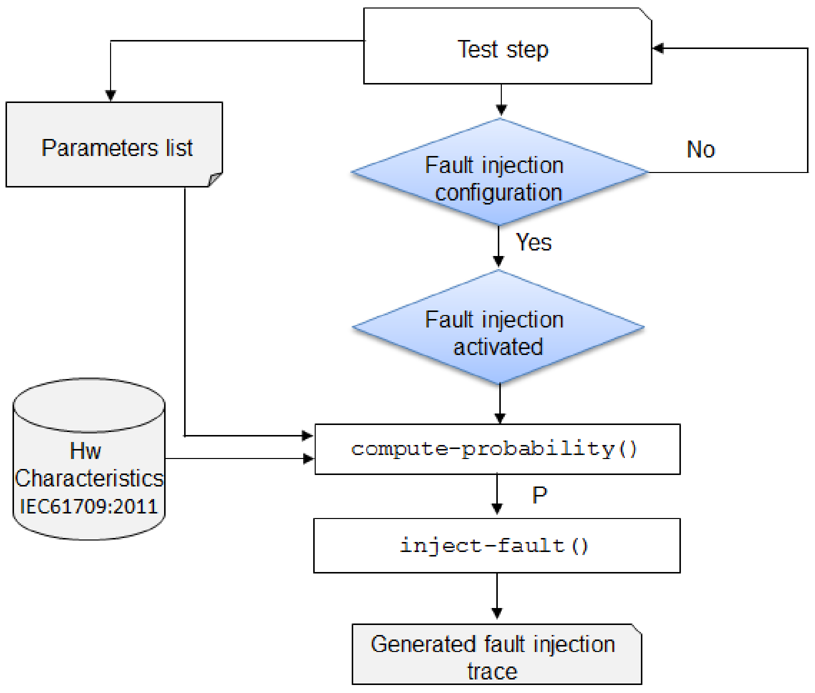

Test step
Parameters list
Fault injection configuration
No
Yes
Fault injection activated
Hw Characteristics IEC61709:2011
compute-probability()
P
inject-fault()
Generated fault injection trace

Fig. 5: Fault injection strategy

The fault injector also takes as input a list of fault configurations which allow to identify the different fault injection configurations. These configurations list is defined depending on the target application requirements and the test objectives. For instance, this list contains the different target memory regions (base address, size) where faults will be injected to avoid unused memory regions and to focus only on allocated regions.

The fault injection strategy is used for two application modes:

- System failure rate prediction
- Evaluate the system robustness

A. *System failure rate prediction*

To measure system failure rate, fault injection is driven by the probability P of occurence of fault for each configuration case. The 'compute-probability' method evaluates the probability P using the reliability model depending on the target hardware component characteristics. By using this probability, the fault injector will take decision of injecting a fault or not.

The HW/SW system is simulated several times and, for each test, the following set of data are collected:

- The test interval length and the number of failures observed in this interval
- Target area tested during test interval
- Date on which each fault is injected and the used application mode

B. *System robustness evaluation*

In contrast with system failure rate prediction, for system robustness study, we want a deterministic injection strategy. In this case the simulation is only driven by the test case scenario. The test case defines the relevant points of fault injection and activates the fault injector module at the appropriate time. Besides, the computed probability doesn't condition the fault injection but it is used to evaluate the criticality of the injected fault. The simulation trace contain the fault probability information, as an annotation, to be used as an input for reliability and robustness quantification task.

VII. Experimentation

To evaluate the effects of hardware failures on the application behavior and evaluate the dependability of the system, we implement the hardware fault model and the fault injection mechanisms by extending the UNISIM-VP [4] virtualization environment. UNISIM-VP (Fig. 6) is an open-source (BSD licensing) simulation environment based on systemC standard [10]. UNISIM-VP provides full system structural computer architecture simulators of electronic boards and system-on-chip (SoC) using a processor instruction set interpreter. The test cases are generated by MaTeLo tool [5]. In MaTeLo, these test cases represent a large number of real-life situations: their execution allows a user to obtain an experimental measurement of the system's operational reliability.

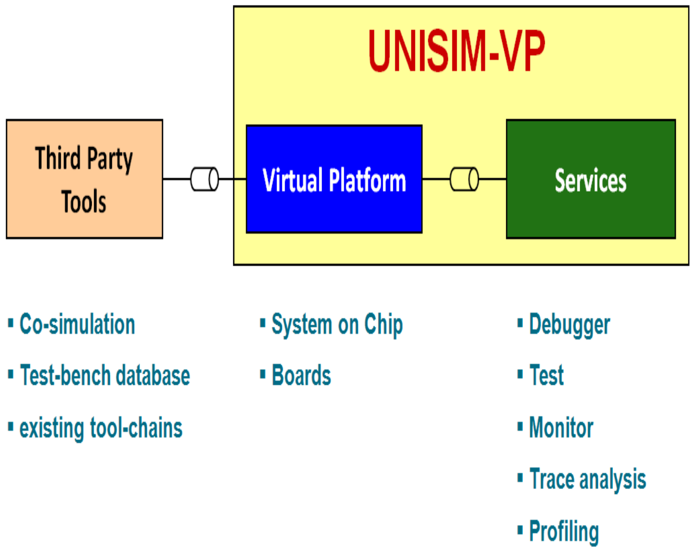


Fig. 6: UNISIM-VP abstract view

The proposed framework has been implemented and tested using a simulator of the freescale micro-controller MC9S12XEP100 and we analyzed the impact of faults on WINDSHIELD WIPER provided by CONTINENTAL as part of EQUITAS project. The perimeter is the complete function: from the action of the driver on the stalk switch until the real wiping of the windshield.

References

[1] H.Lochner, D. Hahn, S. Straube, Complexity, quality and robustness - the challenges of tomorrow's automotive electronics, DATE 2012

[2] A. Bonso, P. Prinetto, "Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation", Frontiers in Electronic Testing, Volume 23, 2003.

[3] August, D. and al., UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development, Computer Architecture Letters, 2007, Feb, volume 6, number 2.

[4] [Online]. Available: http://www.unisim-vp.org

[5] [Online]. Available: http://www.all4tec.net/MaTeLo/homematelo.html

[6] IEC 61709:2011 Electric components-Reliability-Reference conditions for failure rates and stress models for conversion (2nd ed.). Geneva: International Electrotechnical Commission.

[7] H. Ziade, R. Ayoubi, and R. Velazco, A survey on Fault Injection Techniques. The International Arab Journal of Information Technology, Vol.1, No.2, July 2004.

[8] E. Jenn, M. Rimen, J. Ohlsson, J. Karlsson, and J. Arlat, Design Guidelines of a VHDL-Based Simulation Tool for the Validation of Fault Tolerance, in Proceedings of 1st ESPRIT Basic Research Project PDCS-2 Open Workshop, LAAS/CNRS, Toulouse, pp. 461-483, Septembre 1993.

[9] IEC61709:2011 Electric components-reliability-reference conditions for failure rates and stress models for conversion (2nd ed). Geneva: International Electrotechnical Commission.

[10] IEEE-1666 Open SystemC Language Reference Manual. [Online]. Available: http://standards.ieee.org/getieee/1666/index.html

Testing the Resilience of Fail-Operational Systems Early On with Non-Intrusive Data Seeding

Joachim Fröhlich
Siemens AG
Otto Hahn Ring 6
81739 München, Germany

Jelena Frtunikj
fortiss GmbH
Guerickestrasse 25
80805 München, Germany

Alois Knoll
Technische Universität München
Boltzmannstrasse 3
85748 Garching, Germany

I. Fail-Operational Systems are Resilient

Fail-operational systems are resilient systems. Fault patterns and the system structure, e.g., the degree of redundancy, the independence of fault regions and the availability of resources, determine the elasticity of a fail-operational system. Faults can be considered to deform a fail-operational system temporarily or permanently. Faults that occur temporarily or intermittently give systems reversible resilience. Permanent faults and component failures give systems irreversible resilience. When working under real-time constraints, the pressure on a fail-operational system increases. Mechanisms for detection, evaluation and handling of faults must attempt to promptly reshape the system when deformations occur.

II. Hard and Soft System Deformations

The safety analysis and the corresponding fault model of a fail-operational system define the HW and SW faults that it must detect and handle. HW faults, however, do not occur deterministically; they arise stochastically, which can be problematic for new HW for which there is little empirical data. SW faults, on the other hand, occur with certainty. The problem with SW faults is to find the faulty program execution paths and critical combinations of program state and data for complex systems in different situations. Under these conditions, design, implementation and execution of repeatable tests that stimulate the system with faults (fault-injection tests) are difficult and expensive. Tests are considerably facilitated if they can access system internals, that is, can read and write data as it flows through the system as well as access data quality indicators (time and value quality).

III. Tests in Virtual and Real Environments

From technical and economic points of view it is advisable to start as early as possible with the evaluation of individual and integrated systems. It is profitable and advisable to test their capabilities to handle faults promptly and correctly even prior to the availability of actual system HW through the use of virtual environments like HW abstracting test beds. Hence, tests evaluating resilience properties of a system have to be designed and maintained as cross-platform and cross-phase regression tests. To this end, a safety-critical system must be modular and portable. In particular, a fail-operational, real-time system must enable fault-injection tests free of side-effects in order to avoid undesirable functional and temporal distortions of the system under test (SUT). Only then can these tests provide reliable statements about the fault-elasticity of a system that are traceable between virtual and real environments.

IV. Functions and Form of Testable Systems

To obtain definite statements about the effectiveness of fault handling mechanisms from test runs, tests must seed data, including HW and SW faults, at precise time points and system locations. More specifically, the SUT must enable non-intrusive monitoring and manipulation of signal data, state data and data quality indicators. We therefore assume the SUT to basically have the following properties (Fig. 1):

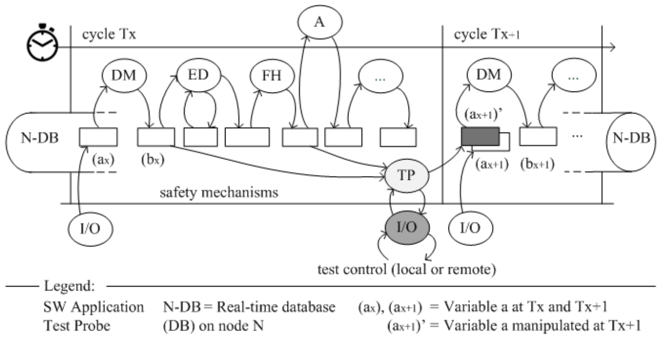

cycle Tx
cycle Tx+1
A
DM
ED
FH
...
DM
...
N-DB
(ax)
(bx)
(ax+1)'
(ax+1)
(bx+1)
...
N-DB
TP
safety mechanisms
I/O
I/O
I/O
test control (local or remote)
Legend:
SW Application
Test Probe
N-DB = Real-time database (DB) on node N
(ax), (ax+1) = Variable a at Tx and Tx+1
(ax+1)' = Variable a manipulated at Tx+1

Fig. 1. Data flow in a time-triggered system node with a built-in test probe

(1) *Time-triggered architectures* (TTA) [3] behave deterministically because systems control events and not vice versa as in event-triggered architectures. In each cycle the system sequentially executes safety operations, like data monitoring (DM), error detection (ED) and fault handling (FH).

(2) *Databases* continually capture, for each node and cycle, flows of signal data, state data and data quality indicators.

(3) All system nodes contain *built-in test probes* (Fig. 1, Fig. 2, TP). TP operations are always scheduled at the end of each cycle. In this position between adjacent cycles, a TP can (i) monitor data accumulated in the N-DB during the last cycle and (ii) manipulate data for the next cycle.

(4) TPs use exclusively reserved resources, including time slots, memory areas and communication links. Other modules cannot use TP resources, even when a TP is deactivated; otherwise, a TP would be intrusive because the SUT would behave differently from the final system.

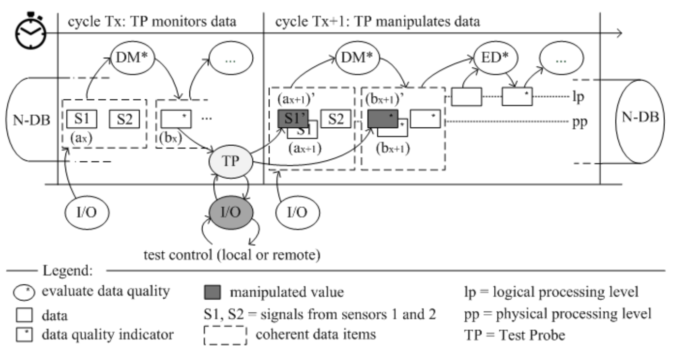

cycle Tx: TP monitors data
cycle Tx+1: TP manipulates data
N-DB
DM*
...
S1
S2
(ax)
(bx)
TP
(ax+1)'
S1'
S1
(ax+1)
(bx+1)'
(bx+1)
DM*
ED*
lp
pp
N-DB
I/O
I/O
I/O
test control (local or remote)
Legend:
evaluate data quality
data
data quality indicator
manipulated value
S1, S2 = signals from sensors 1 and 2
coherent data items
lp = logical processing level
pp = physical processing level
TP = Test Probe

Fig. 2. Manipulate data (ax+1)' or data quality indicator (bx+1)'

V. System of Non-Intrusive Test Probes

Remote tests, local tests, or both combined, control a test probe (TP). Remote tests require a point-to-multipoint connection between one common, central test processor at one end and several test probes at the other ends. Test programs on the central test processor monitor and control nodes system-wide, while test programs on a local node operate within the node. Probe instructions are location transparent: a test probe does not know where the program that issues instructions runs.

Probe instructions manipulate, monitor and check data of the probe-containing node with a single cycle delay. If a remote test program on a central test processor controls one or more test probes, then the test reaction delay for a round trip of a test control loop takes 4 cycles minimum: (i) Test probes read and send values to the central processor in cycle Tx, (ii) the central processor evaluates received values, takes decisions and instructs test probes in cycle Tx+1, (iii) test probes follow the instructions received in cycle Tx+2 which are (iv) effective 1 cycle later in cycle Tx+3. Throughout a test, probes and the central test processor monitor node vitality and test plausibility. Because of test probes running synchronously with the node, tests provide cycle-accurate results in virtual environments and real environments for target HW in the lab and in the field.

VI. Testing System Resilience

For demonstration purposes we test a system of three nodes, instances of which are parts of larger systems that can be built with RACE[1] [1], [4]. Two redundant sensor nodes in the system periphery provide signal data to a central processing node. The test (Algorithm 1 written in ALFHA[2] [2]) checks the elasticity of the central node if a sensor fails temporarily (reversible resilience) or permanently (irreversible resilience).

The test injects (seeds) faults into one of the input channels that connects the central node with the sensors (Fig. 2, (ax+1)') after all nodes are up and run normally (lines 11-12). Alternatively, the test can intervene in the data flow in the central node by manipulating data quality indicators that signify value and time quality of the sensor data (Fig. 2, (bx+1)'). Algorithm 1 allows both approaches (line 18) because the location where the test seeds data is a parameter, as is the fault duration (lines

[1]Robust and reliant Automotive Computing environment for future Ecars, www.projekt-race.de/en

[2]Assertion Language for Fault-Hypothesis Arguments

Algorithm 1 Masking sensor fault on signal receiver

```
 1: TEST Masking sensor fault WITH
 2:     $N$, $N1$, $N2$, // Central node N process signals from sensors N1, N2
 3:     $F$, // N executes system function F
 4:     $L$, $V$, // Location L to inject V in N
 5:     $C$, $D$, // Injection instant (cycle C) and duration (number of cycles D)
 6:     $T$, // Tolerance across 2 succeeding values V
 7:     $P$ // Length of node period P in milliseconds
 8: EXPECT Function gets data from redundant sensor
 9: SYSTEM PERIOD $P$ // Cycle length
10: TIME BOUND 1000 // Obtain definite verdicts within 1000 ms
11: SETUP Masking sensor fault WITH $N$, $N1$, $N2$
12: CLOCK WHEN $N*.State == eNormalOperation$
13: INVARIANT // N, F are resilient to sensor shocks:
14:     // Uninterrupted, smooth data stream
15:     $N.F.Input == N.F.Input@[-1]$ TOLERANCE $T$
16: BEGIN
17:     // Invariant holds (!!) during fault injection ...
18:     $[\ C : < +D\ ]$ !! $N.L = V$ // ... from C to C+D-1
19: END
```

4-5). In any case, the system function that processes the sensor data must always get valid input values (test invariant, line 15). It is not necessary for the test to simulate the environment because, with RACE, nodes can start and run in a neutral mode processing default values. With the steering wheel in neutral position (default), the reversible elasticity of a node executing the steering function can be tested as follows:

$N = CentralNode$, $N1 = Sensor1$, $N2 = Sensor2$, $F = Steering$, $L = In.Sensor1$, $V = InvalidData$, $C = 100$, $D = 2$, $T = 1$, $P = 10$

For testing irreversible elasticity, we extend the fault injection duration D from 2 cycles to, say, 1000 cycles indicating a permanent failure of *Sensor1*. For testing the fault-resilience of the system at various nodes, including the communication links between them, test invariant (line 15) and fault injection (line 18) must refer to different nodes. Then we can change the fault-injection location to, for example, *Sensor2.Out.Steering*, independent of the test invariant still checking the correctness of the data stream at *CentralNode.Steering.Input*.

VII. Resilience Tests Strengthen Safety Cases

Testing fail-operational systems early on with non-intrusive data seeding provides quick and precise feedback in virtual and real environments. Such reliable tests are strong arguments in safety cases of fault-resilient, fail-operational systems.

References

[1] J. Fröhlich and R. Schmid. Architecture for a Hard-Real-Time System Enabling Non-intrusive Tests. In *25th IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, page 24, November 2014.

[2] J. Frtunikj, J. Fröhlich, and A. Knoll. Qualitative Evaluation of Fault Hypotheses with Non-Intrusive Fault Injection. In *5th International Workshop on Software Certification (WoSoCer 2015)*. IEEE, November 2015.

[3] H. Kopetz and G. Bauer. The Time-triggered Architecture. *Proceedings of the IEEE*, 91(1):112–126, January 2003.

[4] S. Sommer, A. Camek, K. Becker, C. Buckl, A. Zirkler, L. Fiege, M. Armbruster, G. Spiegelberg, and A. Knoll. RACE: A Centralized Platform Computer Based Architecture for Automotive Applications. In *Vehicular Electronics Conference and the International Electric Vehicle Conference (VEC/IEVC)*. IEEE, October 2013.

A HW-dependent Software Model for Cross-Layer Fault Analysis in Embedded Systems

Christian Bartsch, Nico Rödel, Carlos Villarraga, Dominik Stoffel and Wolfgang Kunz
Department of Electrical & Computer Engineering
University of Kaiserslautern, Germany

***Abstract*—With the advent of new microelectronic fabrication technologies hardware devices are emerging with an intrinsically higher susceptibility to faults than previous devices. This leads to a substantially lower degree of reliability and demands further improvements of error detection methods. However, any attempt to cover all errors for all theoretically possible scenarios that a system might be used in can easily lead to excessive costs. Instead, an application-dependent approach should be taken, i.e., strategies for test and error resilience must target only those errors that can actually have an effect in the situations in which the hardware is actually used.**

In this paper, we therefore propose a method to inject faults into hardware and to formally analyze their effects on the software behavior. We describe how this analysis can be implemented based on a recently proposed hardware-dependent software model called *program netlist*. We show how program netlists can be extended to formally model the behavior of a program in the event of one or more hardware faults. First experimental results are presented to demonstrate the feasibility of our approach.

I. Introduction

In the design of Systems-on-Chip and Embedded Systems measures for increasing error resilience can benefit from an application-dependent approach when determining good trade-offs between effectiveness and costs. Only those errors should be targeted that can actually have an effect in the situations in which the hardware is really used. These situations are defined by the software. Fortunately, as a result of the application-specific nature of embedded systems, most parts of the software do not undergo major changes during the system's lifetime. This is true in particular for low-level software components controlling the communication between the application software and the hardware, implementing important functions for chip management and, not rarely, replacing traditionally hardware-implemented control functions of the system. When taking measures for increasing the system's reliability with respect to HW faults, it seems wise to take this SW into account since different HW faults may have different relevance in their effects on this software layer and, thus, on the entire system.

Fault injection is a well-known technique to evaluate the fault tolerance of a component or system against specific faults [1], [2]. While the standard application for fault injection is to determine the coverage of implemented resilience techniques, our work has a slightly different focus and uses fault injection to increase the efficiency of resilience techniques by determining which faults should be covered.

For this purpose, a computational model, called *program netlist* [3], recently developed for hardware-dependent software verification, has been extended to formally model the effects of hardware faults on the software. Under all possible runs of the software the effect of an injected fault on the program state is precisely determined, including corner cases which are difficult to find for non-formal approaches like simulation. The proposed method does not only enable the modeling of simple faults like single bit flips but also of more complex fault scenarios consisting of multiple faults occurring at different points in time. Such complex fault scenarios may result from practical observations of test engineers or from a preceding analysis, as described in Section IV-A.

In cases where the protection of specific variables against faults is considered critical and should be increased, we propose to analyze the data dependencies between assembler instructions. In this way, it is possible to understand the possible root causes of faults and the possible fault propagations through the program so that appropriate countermeasures can be taken.

The paper is structured as follows. In Section II we briefly discuss already published work related to our topic. Then, we explain in Section III how our model is generated and how it was extended to model faulty program behavior. In Section IV we present the proposed analysis of the modeled effects as well as the analysis of data dependencies. In Section V experimental results are presented and discussed. This is followed by a conclusion where we summarize our work and point out further applications.

II. Related Work

Most existing approaches for HW/SW cross-layer fault analysis, such as [4], [5], are based on simulation. It is in the nature of simulation-based approaches that full confidence can never be gained on the absence of errors. For the same reason a complete understanding on how a fault can propagate and how it can affect the program execution is not possible. The proposed formal approach can provide this confidence and an improved understanding of possible fault propagations by exploring all possible program runs under a given fault assumption.

Our approach is therefore also useful to formally certify the effectiveness of resilience measures or to prove that one resilience measure is more effective than another based on some metrics. In [6] related goals are pursued and specific code transformations were proposed to increase the robustness of software against hardware faults. However, the verification in [6] relies on a simulation-based method. Verifying the effectiveness of such approaches in a formal way can increase the confidence about the applied resilience measures and allows for formulating safety guarantees for the system.

There is also previous work in which formal techniques were used to analyze the effects of HW faults on the system behavior. In [7], for example, model checking is used to prove that specific fault tolerance properties hold. A use case of this technique was presented in [8]. In order to perform the fault analysis a labeled transition system (LTS) representing the considered system has to be generated. Model checking was also used in [9], where the fault tolerance of a startup algorithm for a time-triggered architecture had been proven. Like in [8] a manual conversion of the HW/SW system into a highly abstracted state transition system is conducted. Such approaches are promising when a manual translation of the concrete system into a highly abstract model is doable. However, in some industrial settings a higher degree of automation is required. Moreover, the standard fault models for HW implementations like stuck-at faults and bit flips do not have a direct correspondence at the abstract level. For the same reason fault effects on the detailed I/O-behavior are difficult to model at the abstract level such that their analysis may be difficult or even completely impossible.

In [10] a formal framework is proposed which is able to analyze the effects of transient HW faults on the program behavior. The proposed framework enumerates the effects of an error for every possible error location, e.g. every register in the register file, at one or more instructions of the program. However, it cannot handle undefined inputs to the program, so that concrete input values have to be chosen. In addition to that the assembly program under consideration has to be converted into a custom-built assembly language supporting only a small set of instructions. Analyzing how a fault affects the temporal behavior of a program is therefore very difficult. The same difficulties occur when trying to derive low-cost resilience measures by exploiting the knowledge about the used instruction set architecture and possible program states. Another drawback is that the framework operates on the word-level, leading to over-approximation and incomplete knowledge on the program's control flow. This may lead to complexity problems (e.g. by introducing infinite loops) as well as to false alarms when attempting to certify the effectiveness of resilience measures.

Our approach can be seen as complementary to previous work that has evaluated the effects of HW faults on the architectural processor state, such as [11]. There, it is elaborated on how intermittent HW faults on the RT level affect the behavior of processor components, including program-visible components like the register file. Knowledge about how physical defects propagate through the layers can be used to develop realistic fault models on the architectural level and provide a basis for methods like the one proposed in this work. Another promising approach developing realistic fault models is to derive them from a meta-model [12].

Finally, the output of our analysis can be used as input for techniques that inject faults directly into the software, e.g. [13], to examine their effect in higher software layers than the ones considered here.

III. Program Netlist

The underlying model of the proposed fault analysis method is called *program netlist* (PN) [14]. A PN formally models the behavior of a processor with respect to a specific software program. An important property of this model is that it is of entirely combinational nature so that Boolean reasoning by SAT-solving can be employed for its analysis. This was exploited by [3] to develop an efficient equivalence checker that will be used in the proposed fault analysis, as described in Section IV.

A. Model Generation

The process for the model generation consists of two steps. The first step is to unroll a control flow graph (CFG) representing the software program. The CFG can be obtained by extraction from either machine or assembly code of the program. In our model, each node of the graph represents an individual instruction. The unrolled CFG is called *execution graph* (EXG) and is the basis for the second step, where each node of the EXG is replaced by a corresponding logic block describing the behavior of the processor for this particular instruction. Such a logic block is called *instruction cell* (IC). It has an input and an output which are connected to the preceding and succeeding instruction cell, respectively. The input of an IC represents the current program state, i.e., the values of the program variables in memory and the contents of the CPU registers before the corresponding instruction is executed. Its output represents the next program state, i.e., the situation after the instruction was executed.

The CFG used as the starting point for model generation can be incomplete (e.g., branch targets may be unknown because of indirect addressing), as is often the case when a CFG is generated from a real software program. This incompleteness is acceptable because the missing information is generated during the model generation process. This is done by interleaving the unrolling process with a SAT-based analysis to fill in the missing information. The interleaved analysis also supports a compaction of the model.

```
ADD(const Rm, const Rn, in PS, out PS')
{
    PS' = PS;
    PS'[Rn] = PS[Rn] + PS[Rm];
}
```

Fig. 1. Instruction Cell

An example of an instruction cell template is shown in Fig. 1 by using pseudo code[1]. It depicts the behavior of an ADD instruction of the SuperH2 instruction set architecture (ISA).

[1]Information about bit widths are abstracted in this and the following examples to make them more readable.

As can be seen, the instruction cell needs to know on which registers the operation should be performed. This information is encoded in the specific assembler instruction of a program. The identifiers Rm and Rn in the template will be replaced with the actual register addresses when the instruction cell is instantiated during PN generation. The instantiated instruction cell has only one input, the current program state, and one output, the next program state. The body of the instruction cell basically contains a forwarding of the program state from the input to the output, but with the exception of the register that contains the result of the performed addition. This register is changed according to the ISA specification.

B. Fault Description

When analyzing the behavior of software with respect to hardware faults a model of the system is required which describes for each fault the time at which the fault occurs, how long it lasts and how its logical behavior affects the execution of an instruction. The logical behavior of a hardware fault can be modeled by describing its effects on the program state, i.e. how a faulty instruction execution deviates from the correct one. This can be accomplished by changing the corresponding instruction cell description in an appropriate way and is explained in the next section.

In order to model the temporal behavior of a fault, a cycle-abstract representation of time was used in this work. This reflects the need for a time-abstract view on the program execution in order to handle larger processors with unpredictable execution times. Abstraction was performed in the way that time is represented by the order and position of the instructions in the program. The proposed method, however, is easily adaptable for time-accurate instruction cells that can be created for processors with predictable execution times.

In the case a single hardware fault has an effect on multiple processor instructions, the effects can be modeled by creating a corresponding fault description for every affected instruction cell. Correct modeling of multiple faults and their effects on different instructions is more challenging. As will be elaborated below, this can be achieved by adding additional constants, registers and ports to the fault description of an instruction cell. (In our modeling of PNs, ports represent the memory interface of I/O instructions [14].)

```
ADD_Faults()
{
    -- Fault1
    Fault_Cond += 1;
    if(Fault_Cond.bit(LSB) == 0)
    {
        PS'[Rn].bit(MSB) =
                Fault_Register.bit(MSB);
    }

    -- Fault2
    ...
}
```

Fig. 2. Fault Cell

The example shown in Fig. 2 illustrates the most simple case of a fault description that can be integrated into the description of an instruction cell. In this example, a stuck-at fault is modeled which is activated only every second ADD instruction and which affects the most significant bit (MSB) of the addition. For this purpose the architectural state was extended by two registers: Fault_Cond and Fault_Register. The former has an initialization value of zero while the latter is left uninitialized. The Fault_Cond register is incremented every time the ADD instruction is executed, and the fault becomes active whenever the least significant bit (LSB) of the Fault_Cond register is zero, i.e., on every second incrementation of Fault_Cond. Then, the MSB of the target register is assigned the value of the MSB of the unspecified register Fault_register. In effect, the MSB of the target register is treated like an open input in our formal analysis. This way, both faults, stuck-at-0 and stuck-at-1, can be considered at the same time.

Note that it is possible to describe more than one fault for a particular instruction, as indicated in the lower part of Fig. 2. A fault description with more than one fault can serve two purposes. It can be used to model multiple faults and to examine their combined effect on the program. The second purpose is to model several faults (single or multiple) in the same program netlist, thus avoiding the effort of re-generating the program netlist for every fault to be examined.

In order to support such complex fault descriptions for fault lists with a large number of faults and to separate the activation and deactivation of faults from the computation of the internal processor state, the occurrence of a fault given in the fault list is encoded into the data of an auxiliary memory at a specific location addressed through auxiliary ports. These auxiliary ports do not correspond to variables of the original software but are only used in our computational model to gain better control on the activation conditions.

During fault analysis the values of these ports are set appropriately so that specific faults and combinations of faults can be activated or deactivated. In fact, using this construction, it is sufficient to generate a single PN to analyze the effects of several single faults and/or several multiple faults together with the original fault-free behavior.

```
ADD_Faults(Fault_Port Port)
{
    -- Fault1
    Port.Address = 0xABCD;
    Fault_Cond += 1;
    if((Fault_Cond.bit(LSB) == 1) &&
        (Port.Data == 1))
    {
        PS'[Rn].bit(MSB) =
                Fault_Register.bit(MSB);
    }

    -- Fault2
    ...
}
```

Fig. 3. Fault Cell

In Fig. 3 the code of Fig. 2 was modified such that a memory access was added to the fault description. Now the fault is active only when also the LSB of the read data is equal to 1.

C. Fault Injection

Faults are injected to instruction cells by inserting their description at the end of the corresponding cell. The injected fault changes the behavior of the original instruction cell by either performing additional changes of the program state or by overwriting changes of the fault-free part.

```
ADD(const Rm, const Rn, in PS, out PS')
{
    PS' = PS;
    PS'[Rn] = PS[Rn] + PS[Rm];

    -- Fault1
    Fault_Cond += 1;
    if(Fault_Cond.bit(LSB) == 0)
    {
        PS'[Rn].bit(MSB) =
                Fault_Register.bit(MSB);
    }

    -- Fault2
    ...

    -- Fault3
    ...
}
```

Fig. 4. Fault Cell

In example 4 an instruction cell with several fault injections is shown. By adding both the temporal activation conditions and the descriptions of the logical fault behaviors to the instruction cells yields the advantage that during the model generation process no complex fault injection is required. As described in [14], the PN model generation steps are interleaved with a SAT-based analysis to prune the control space of the program. This analysis is now extended to the instruction cells with their fault descriptions so that all possible fault scenarios are also included into the generated model. Based on the obtained PN faults are simply injected by making value assignments to the addresses of the auxiliary memory.

Note that since the PN represents all fault behaviors of the fault list, it is also possible to perform a global reasoning over all faults or sets of faults. For example, the set of all faults could be determined that lead the program into a specific program state.

Obviously, a PN modeling a large number of possible faults may turn out to be more complex than the corresponding PN for the fault-free case or the PN for only a small subset of these faults. Depending on the complexity of the model it may therefore be advisable to partition the fault list and to analyze each partition in a separate PN.

IV. Fault Analysis

The resulting model can be used to analyze the effects of HW errors on the SW behavior including program states, I/O sequences and the control flow. Two approaches for fault analysis are possible. The first one is to compare the behavior modeled by the PN against an abstract specification using a HW property checker. The second approach is to compare the PN containing faults with its fault-free counterpart. The latter approach is preferred here since it can benefit from sophisticated optimizations used in standard hardware equivalence checking. An appropriate method to check the equivalence of two different PNs was already proposed in [3].

We define two programs to be equivalent iff they produce the same output sequence for any applicable input sequence. Any modification, for example by activated faults, of the original program can be considered a different program. Equivalence checks can be performed on the same PN but with a different set of faults activated in each check. The PN with all faults deactivated can be used as the fault-free reference.

There are two possible outcomes of the equivalence check. The first possibility is that the PNs are equivalent, i.e., the corresponding programs produce the same I/O sequences. In this case, the considered fault has no effect on the program behavior regardless of what values the inputs have. We denote such faults as *application-redundant*. In the other case, if the PNs are not equivalent, this means that they differ in either data or address of one or more I/O accesses, in the number of I/O sequences or a combination of these. In such cases, a subsequent analysis may be used to categorize the error. For example, a simple structural analysis of the two PNs can yield the information on whether the considered fault affects only data or modifies the control flow of the program.

A. Dependency Analysis

In some applications, resilience measures are desirable which do not protect the entire program but only specific functions or instruction sequences inside a function. For example, a loop counter may be considered more critical than some variable within the loop. In order to ensure the correct execution of these critical instructions, however, resilience measures only protecting these particular instructions might not be sufficient. Due to the nature of the program's computation a fault activated during the execution of instructions with low criticality might actually propagate to critical instructions. Therefore, we propose to perform an analysis to determine the data dependencies of critical instructions. This calculation can be done by analyzing the PN and provides a precise description of all dependencies. Note that pure machine code would not be sufficient for this analysis since it yields only incomplete CFGs and therefore would lead to an over-approximation of possible dependencies.

As an example, Fig. 5 shows an excerpt of the results from a dependency analysis. The analysis was performed on the PN of the Traffic Alert and Collision Avoidance System developed by Siemens which is part of the Software-artifact Infrastructure Repository [15]. For demonstration of our analysis, an instruction was selected which delivers the value for a variable that is important for the result calculation of the overall algorithm. The instruction is shown as the bottom node (with address 1432) in Fig. 5. The performed dependency analysis took less than 2 seconds[2] to recursively determine control and data dependencies.

The figure shows an excerpt of all dependencies existing for the considered instruction. These dependencies are extracted from all possible program paths leading to this instruction and are represented in a dependency graph as shown. The depicted nodes represent instructions and their addresses as well as information on what registers or memory locations are read (R) or written (W) by each instruction. The annotation "`R: @R1 (0x0000)`", for example, indicates that the particular instruction reads from the memory address 0x0000 stored in register R1. Similarly,"`W: R1`" means that the particular instruction writes to R1.

As mentioned before, only an excerpt of the dependency analysis is shown. Parts which were removed are indicated by a dashed line. Next to solid lines the dependency type is noted. As summarized in Table I, "Type 0" indicates a direct data dependency where one instruction writes to a register which is used by another. "Type 1" also indicates a data dependency but in this case one that exists through a memory value rather than register content. The last type, "Type 2", indicates that the particular instruction depends on a correctly executed jump or branch instruction.

It is worthwhile noting that the uppermost node (with address 1432) represents a jump instruction which needs the value of a register to calculate the jump target address. Due to the characteristics of the used model all possible target addresses are known, so that not it is possible to trace in the PN both in forward and backward direction to extract the relevant dependencies.

TABLE I
EXPLANATION

Dependency Type	Explanation
Type 0	Data Dependency (Register)
Type 1	Data Dependency (Memory)
Type 2	Control Flow Dependency

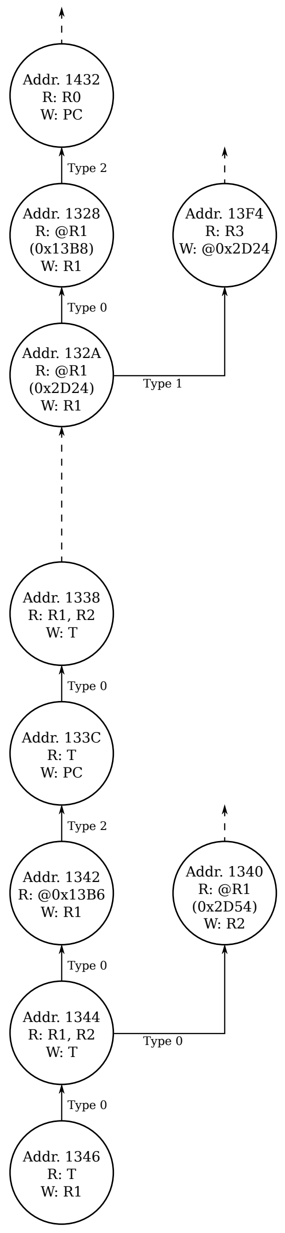


Fig. 5. Dependency Analysis

As can be noted, the paths in the dependency graph of Fig. 5 are not numbered with consecutive instruction addresses. In fact, in our experiments it could be observed that the topology of the computed dependency graph is not identical and not even in a simple relationship with the topology of the program's execution graph. This demonstrates that indeed additional information is obtained from the proposed analysis which may be valuable when designing cost-efficient resilience solutions. Their effectiveness can be certified by proving equivalent behavior of the protected code segment for a given fault list.

V. EXPERIMENTAL RESULTS

Experimental results have been conducted on a HW platform containing a 32-bit processor (Aquarius, SuperH2 instruction set) running two different programs. One is a software-implemented industrial driver for a master node of the automotive protocol LIN. Another is the traffic alert and collision avoidance system (TCAS) [15] developed by Siemens and part of the software-artifact infrastructure repository.

An evaluation on how the fault injection affects the model generation runtime and complexity was conducted. For each considered test program two PNs were generated. The first PN, referred to as fault-free, was generated without injecting any faults, while in the second run different faults were injected into the PN.

Different types of faults such as stuck-at faults and bit flips have been selected manually from a general fault list and have been injected into the HW. In the case of TCAS, the safety critical variable mentioned in Section IV-A was identified in

[2]Measured by using the profiling tool Gprof.

the source code and the proposed dependency analysis was used to create a list of instructions affecting the variable. For a first assessment of our approach, based on this analysis, four instructions were selected. Transient as well as permanent faults were injected. In the case of LIN, permanent faults were injected in all used shift instructions. Such fault scenario can be used to model a faulty shifting unit.

It is worthwhile noting that the performed fault injection process was exhaustive. For all bits used in arithmetic and logic operations as well as in read/write processes from/to the register file or from/to the ports all possible faults of types stuck-at 1, stuck-at 0 and bit flip have been injected. In this way, 353 bits were identified in the case of TCAS and 258 in the case of LIN that can be affected by any of these faults. All faults of the different fault types can be analyzed either independently as single faults or in arbitrary combinations as multiple faults. In this preliminary experiment, we ignored multiple fault scenarios and restricted our analysis to find out which of the injected single faults can possibly have an effect on the program behavior. The analysis was done by performing equivalence checks using the commercial tool OneSpin [16].

Table II shows the number and type of faults injected for each test program. The number of injected faults also represent the number of analyzed single bit faults for the particular fault model. This makes a total of 1059 single bit faults for the TCAS example and 774 single bit faults for the LIN example. We used the GCC compiler to compile the test programs. All experiments were performed on an Intel i7-4790 CPU at 3.6 GHz with 16 GB RAM. The timing measurements were performed using the profiling tool Gprof.

TABLE II
INJECTED FAULTS

Program	Bit Flips	Stuck-at 0	Stuck-at 1
TCAS	353	353	353
LIN	258	258	258

TABLE III
CPU TIMES FOR MODEL GENERATION

Program	CPU time (s.)	
	Fault-Free	Faults Injected
TCAS	10.13	22.72
LIN	76.07	120.99

Table III shows the time required to generate the PNs. It can be observed that fault injection has a significant effect on the runtime of the PN generation process. However, when taking into account that a huge number of different single bit and multiple bit faults are modeled in the PN the increase in runtime seems acceptable.

For each generated PN Table IV shows how many instructions it contains. It can be observed that the fault injected PN generated for the TCAS program is only 5 instructions or less than 1% larger than the fault-free PN. Obviously, the faults that have been injected in most cases do not have an effect on the control flow of the program, although one of the four selected instructions directly affects a branch instruction. We believe that the reason for that is that the program behavior is mostly data-driven such that most branches of the program are already included in the fault-free PN. For the LIN program the fault injected PN is larger by 320 instructions (17%) compared to its original version. As a results of the injected faults the program was able to take program paths which were previously unreachable, adding several new instructions to the PN.

TABLE IV
NUMBER OF INSTRUCTIONS OF MODELS

Program	# of Instructions	
	Fault-Free	Faults Injected
TCAS	655	660
LIN	1862	2182

Analyzing the results of the proposed fault analysis can yield important insights into the effects of the considered faults at the software level. For example, in the case of TCAS only 394 injected single bit faults proved to actually have an effect on the value of the selected variable.

Taking into account the entire I/O behavior of the system we were able to prove application redundancy for 561 faults w.r.t. TCAS and 552 faults w.r.t. LIN. These fairly large numbers demonstrate the value of the proposed analysis and suggest that the effect of HW faults at the SW level varies widely. Beyond application redundancy also other fault scenarios of interest to the user can be explored. For example, for certain faults of the LIN bus we could observe that the LIN node was virtually disconnected from the bus.

VI. Conclusion and Future Work

The paper has demonstrated the feasibility of analyzing the effects of HW faults at the SW level by using formal methods. Formal methods provide the advantage that they can actually *certify* the absence of errors for a given application or the effectiveness of fault resilience measures. This can form the basis for developing test strategies, for example by exploiting the knowledge about application-redundant faults, as well as for designing cost-efficient and effective fault resilience mechanisms both at the HW and the SW level. Such applications of the proposed techniques are subject to our future work.

References

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

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

[3] C. Villarraga, B. Schmidt, C. Bartsch, J. Bormann, D. Stoffel, and W. Kunz, “An equivalence checker for hardware-dependent software,” in *11. ACM-IEEE International Conference on Formal Methods and Models for Codesign*, 2013, pp. 119–128.

[4] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. 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*, ser. ASPLOS XIII, 2008, pp. 265 – 276.

[5] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, “Characterizing the impact of intermittent hardware faults on programs,” *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 297–310, March 2015.

[6] A. Sharma, J. Sloan, L. Wanner, S. Elmalaki, M. Srivastava, and P. Gupta, “Towards analyzing and improving robustness of software applications to intermittent and permanent faults in hardware,” in *International Conference on Computer Design*, October 2013, pp. 435 – 438.

[7] C. Bernardeschi, A. Fantechi, and S. Gnesi, “Model checking fault tolerant systems,” *Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 251 – 275, 2002.

[8] ——, “Formal validation of the guards inter-consistency mechanism,” in *Computer Safety, Reliability and Security*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, vol. 1698, pp. 420–430.

[9] W. Steiner, J. Rushby, M. Sorea, and H. Pfeifer, “Model checking a fault-tolerant startup algorithm: from design exploration to exhaustive fault simulation,” in *Dependable Systems and Networks, 2004 International Conference on*, June 2004, pp. 189–198.

[10] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, “Symplfied: Symbolic program-level fault injection and error detection framework,” *IEEE Transactions on Computers*, vol. 62, no. 11, pp. 2292 – 2307, Nov 2013.

[11] J. Gracia-Moran, J. Baraza-Calvo, D. Gil-Tomas, L. Saiz-Adalid, and P. Gil-Vicente, “Effects of intermittent faults on the reliability of a reduced instruction set computing (risc) microprocessor,” *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 144–153, March 2014.

[12] M. Schwarz, M. Chaari, B.-A. Tabacaru, and W. Ecker, “A meta-model-based approach for semantic fault modeling on multiple abstraction levels,” in *Design and Verification Conference and Exhibition Europe*, November 2015.

[13] D. Larsson and R. Haehnle, “Symbolic fault injection,” in *Proc. 4th International Verification Workshop (Verify) in connection with CADE-21*, vol. 259, 2007, pp. 85 – 103.

[14] B. Schmidt, C. Villarraga, T. Fehmel, J. Bormann, M. Wedler, M. Nguyen, D. Stoffel, and W. Kunz, “A new formal verification approach for hardware-dependent embedded system software,” *IPSJ Transactions on System LSI Design Methodology (Special Issue on ASPDAC-2013)*, vol. 6, pp. 135–145, 2013.

[15] Software-artifact infrastructure repository. http://sir.unl.edu. Accessed: 2015-09-01.

[16] Onespin. http://www.onespin-solutions.com/.

Component Fault Localization using Built-In Current Sensors for Error Resilient Computation

***Abstract*—Identification of the exact fault within a system helps in computing its faulty Boolean function, which thereby facilitates the usage of the faulty component within the system, in conjunction with a software wrapper that corrects the error. Using principles of electromagnetic circuit theory, we analytically show that if we use multiple on-chip current sensors, the average current drawn by the faulty components during the system testing, can be used to identify the exact manifested fault. We validate the proposed technique by modeling and simulating the power-grid using SPICE. The proposed technique, when applied to standard components found inside computers, helped to localize almost all the logically equivalent faults.**

I. Introduction and Background

In principle, logic-based fault localization methods [1]–[3] are incapable of distinguishing logically-equivalent faults (LEFs). In this paper, we address this issue by localizing logically equivalent faults, and we argue that this can help in computing the Boolean function of the faulty component, and this helps in correcting the error in software by writing a wrapper Boolean correction function, as shown below:

$$F^{'} = F + e(f) \tag{1}$$

where F the component's fault-free Boolean function and $F^{'}$ is the error resilient Boolean function of the component in the presence of fault f, and $e(f)$ is the error correction function for fault f. Practically, the operating system (OS) accomplishes this through rewriting the binary, as explained in [4]. In order to accomplish that, the OS should localize the fault and apply suitable correction function, as shown in Equation (1) to accomplish error resiliency.

Although two faults are logically-equivalent, they can produce different levels of switching activity on different wires inside the component. This switching activity information is not available on the logical connections outside the component (primary inputs and primary outputs), and thus needs to be gathered from *the power grid*, that supplies current to all the switching gates inside the chip [6]. We use built-in current sensors for measuring the power information from the power-grid, and use this information to localize faults. This paper proves the feasibility of measuring power information that distinguishes LEFs using electromagnetic circuit theory and validates the same using SPICE simulations. The next section explains the *principle of charge conservation*, the proof and how it is used to localize faults.

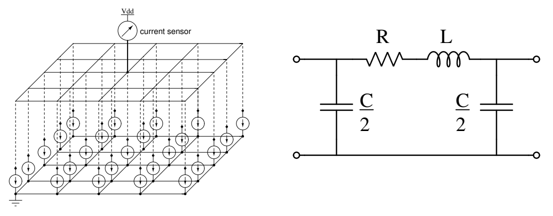
II. The Core Idea

The component circuit is assumed to be in steady state before launching the test pattern, and the circuit settles in a steady state after launching the pattern. We model the power grid as an RLC mesh network using the π-model, and each gate in the digital circuit is modeled as a current sink, as shown in Figure 1.

A. Analytical proof

From Gauss's law, we know that

$$\nabla \bullet (\vec{\mathbf{J}}) = -\frac{\partial \rho_v}{\partial t} \tag{2}$$



(a) Model of the power-grid along with the component modeled as current sinks (b) π-model of each unit in the RLC mesh

Fig. 1. Modeling of the component along with the power grid, where each gate is modeled as a current sink and the power grid as an RLC network

, where $\vec{\mathbf{J}}$ is the current-density vector and ρ_v is the volume charge density. We assume that the power-grid was in steady-state before and after the application of the test-pattern pair. According to Equation 2, this means that the current provided by the power grid to the component, leads to time rate of change of the *charge* in the system consisting of the power-grid and the circuit[1]. Hence, the current provided by the supply pins leads to time rate of change of charge in the circuit, which translates to charging and discharging of capacitors inside the component. Thus, by this principle of charge conservation, the net charge provided by the supply pin is equal to the net change in charge in the digital circuit. If we imagine an enclosed surface around the power grid of volume V, and integrate over this volume, this leads to the following equation,

$$\frac{\partial}{\partial t}\int_V \rho_v dV = \int_V -\nabla \bullet (\vec{\mathbf{J}}) dV \tag{3}$$

which implies

$$\frac{\partial}{\partial t}\int_V \rho_v dV = -\int_{\partial V} JdA = \int_{-\partial V} JdA \tag{4}$$

which implies that the rate of change of total amount of charge within the volume V is equal to the amount of current flowing inwards across the boundary of the volume V (which is equal to the current provided by the supply pin). Now,

$$\rho_v = \rho_v^{power-grid} + \rho_v^{circuit} \tag{5}$$

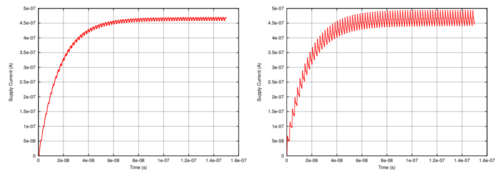
which implies

$$\frac{\partial}{\partial t}\int_V \rho_v dV = \frac{\partial}{\partial t}\int_V \rho_v^{power-grid} dV + \frac{\partial}{\partial t}\int_V \rho_v^{circuit} dV \tag{6}$$

Based on the two assumptions already stated, $\frac{\partial}{\partial t}\int_V \rho_v^{power-grid} dV \approx 0$, hence Equation 3 transforms to

$$\frac{\partial}{\partial t}\int_V \rho_v^{circuit} dV = \int_{-\partial V} JdA \tag{7}$$

[1]under the assumption that the power grid capacitors have some fixed stored charge independent of the test pattern applied, and the power grid inductors carry zero-current



(a) A toggle at the corner of die, diagonally opposite to that of the supply pin (b) A toggle at centre of die

Fig. 2. SPICE simulation of current drawn from the supply pin that is at the corner of a 100x100 power grid (10,000 nodes).

which implies that the rate of change of total amount of charge within the digital circuit is equal to the amount of current provided by the supply pin. This is what we refer to as the *principle of charge conservation*. Let us assume the circuit is already initialized with the first test pattern at $t = 0$ and the second test pattern at this instant. If the clock cycle time be T, then applying time average of Equation 7 leads to

$$\int_{t=0}^{t=T} \frac{\partial}{\partial t}(\int_V \rho_v^{circuit} dV)dt = \int_{t=0}^{t=T} (\int_{-\partial V} JdA)dt \qquad (8)$$

which implies that *the average current drawn by the circuit during the launch-to-capture window is equal to the average current provided by the supply pin during the same window*. **Thus, it is analytically proved using the principle of charge conservation that average current consumed due to switching of gates inside the digital circuit (independent of their positions in the chip layout) is equal to the average current drawn by the supply pin during the launch-to-capture window.** The validation of this principle through circuit simulation, and the results obtained on component benchmarks are provided in the next section.

III. Experimental Setup and Results

Predictive technology [5] is used to model the power-grid at $22nm$ and simulate it in SPICE. We show the proof-of-concept for single supply pin, because more supply pins actually increase the localization resolution of the proposed technique. The following experiment is performed to validate the circuit theory proof shown in the previous section, that independent of the position in the layout, then average current consumption for a toggle is the same: the power grid is simulated in SPICE and the waveforms for the current drawn by the supply pin for two different test cases as shown in Figures 2(a) and 2(b) respectively. From these waveforms, the following observations can be made: (1) The initial part of the waveform grows with time. This is because the initial condition is such that the all the power grid capacitors have zero charge. As a result, it takes time for the power-grid to get charged upto the full-swing supply voltage; (2) After the power-grid reaches full-swing supply voltage, the current waveform in both the cases fluctuates around $0.465\mu A$, which is the average current per single toggle from our calculations; and (3) It can also be observed that, after the power-grid reaches steady-state, even in a single clock cycle (which is the launch-to-capture window), the average current is $0.465\mu A$.

A diagnostic pattern-pair generation algorithm is used to identify pairs of patterns, that distinguish between logically equivalent faults (LEFs), using this current information, obtained from the sensors. Due to lack of space, we skip the detailed discussion of the diagnostic pattern-pair generation algorithm. The effectiveness of the proposed technique is shown for a logically equivalent fault class in

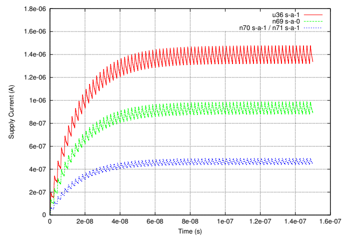


Fig. 3. **Relative difference** in current consumed for LEFs *u36 s-a-1* (**in red**), *n69 s-a-0* (**in green**) and *n71 s-a-1* (**in blue**) in `comparator` benchmark, measured using SPICE

TABLE I
EFFECTIVENESS OF *the proposed technique*

Component	$\#LEF$s	$\#LEF$s localized	% Localization
comparator	82	82	100
BCD-binary converter	49	49	100
resource-arbiter	264	248	94
sequence-identifier	2652	2652	100
sensor-interface	645	645	100
16-bit Integer ALU	5858	5858	100
64-bit FPU	4627	4211	91

`comparator` benchmark, in Figure 3. Further, the results obtained by applying the proposed technique on the LEF sets of several computer component benchmarks are shown in Table I. It can be seen that, in all the cases the localization is $>90\%$.

IV. Conclusions and Future Work

We have proposed a technique for identifying exact fault in a faulty component identified during system testing. The proposed technique is analytically proved using electromagnetic theory, and also validated the same using SPICE simulations. This translates to fault resilient computation, by using the faulty component and correcting the error in software. There are a few cases, where the localization success is not 100%. The localization of these *hard-to-localize* faults is the scope of future investigation. Although, for the sake of experimentation, we have validated our technique for single stuck faults, the technique does not restrict itself to the same, and is equally applicable for bridges, opens and multiple faults. There has been work in the past to avoid faulty components and compute with correctly working components, however at a degraded performance [4]. We strongly believe that our technique , since it does not avoid the faulty components, improves the performance over [4]. The experimentation of the same, is currently under investigation.

References

[1] M. Abramovici, “A Hierarchical, Path-Oriented Approach to Fault Diagnosis in Modular Combinational Circuits”, IEEE Transactions on Computers, Vol. C-31, No. 7, 1982, pp. 672-677.
[2] M. Abramovici, M Breuer and A Friedman. “Digital Systems Testing and Testable Design”, Wiley Publishers, 1994.
[3] I. Pomeranz, S. M. Reddy. “ Same/Different Fault Dictionary: An Extended Pass/Fail Fault Dictionary with Improved Diagnostic Resolution”, Design, Automation and Test in Europe, IEEE, 2008, pp.1474-1479
[4] K. Raghavan and V. Kamakoti, “ROSY: recovering processor and memory systems from hard errors”, ACM SIGOPS Operating Systems Review, Vol. 45, No. 3, 2012, pp. 82-84.
[5] Predictive Technology Model, URL:http://ptm.asu.edu/
[6] S. Potluri et al., ”PinPoint: An algorithm for enhancing diagnostic resolution using capture cycle power information,” European Test Symposium, IEEE, 2013, pp. 1, 2013.

Efficient Fault Emulation through Splitting Combinational and Sequential Fault Propagation

[Extended Abstract]

Ralph Nyberg*, Johann Heyszl* and Georg Sigl‡

*Fraunhofer Institute AISEC, Munich, Germany, Email: ralph.nyberg@aisec.fraunhofer.de
‡Technische Universität München, EI SEC, Munich, Germany, Email: sigl@tum.de

***Abstract*—We present a new strategy for fast fault emulation of transient faults in sequential circuits used to perform fault analysis during design verification. We split the problem of combinational fault modeling into two steps: 1) fault propagation in combinational logic processed by a software approach and 2) sequential fault propagation using a fast FPGA based fault emulator. In this way, we are able to inject faults in sequential as well as combinational cells. Compared to existing work, our method can be applied to larger circuits and it performs the demonstrated fault analysis in a micro-controller design more than two times faster.**

I. Introduction and Problem Statement

Transient faults in digital circuits may occur infrequently and temporarily at runtime caused by, e.g., alpha radiation or other radiation sources. Also fault attacks, where attackers try to compromise security devices by injecting faults deliberately with e.g. a laser or electromagnetic waves, result in transient faults. This is a major concern for designers of security and safety-critical circuits. In order to check proper circuit behavior in the present of faults and hindering attackers from gaining advantages out of fault attacks, safety-critical and security devices utilize fault countermeasures. The effectiveness of fault countermeasures has to be verified by performing fault analysis during a limited time frame for verification in the development cycle of the circuit, where faults are injected into a Device Under Test (DUT) at different locations and for different timings. In this way, vulnerable circuit parts are identified and the effectiveness of fault countermeasures utilized in safety-critical and security devices is validated, helping designers to improve security and safety-critical designs.

FPGA-based fault emulation is the fasted method to perform extensive fault analysis. It is three to five orders of magnitude faster [1], [2], [3] than simulation and software-based symbolic approaches, e.g. [4] and [5]. This high performance provided by fault emulation allows to benefit as much as possible from limited verification times during design. FPGA-based fault emulation environments alter a DUT, e.g. by utilizing the circuit instrumentation technique [2], [3], [6], [7], in order to provide fault injection capability based on a fault model. The instrumented DUT is then synthesized onto an FPGA. In order to support fault injection into combinational or sequential cells, at least one additional flip-flop (FF) for controlling the fault injection, some combinational cells for implementing the fault model and additional routing on the FPGA are required for each instrumented cell. Digital designs usually comprise about five to ten times more combinational cells than sequential cells. By considering fault injection in sequential cells only, as in [2], [8] and [6], the hardware overhead is kept in check but faults cannot be injected into combinational cells, and, as a result, the vulnerability to faults of the combinational logic cannot be analyzed. Fault emulation environments supporting fault injection in combinational cells, e.g. [7] and [3], demand a lot of hardware resources on the FPGA. Additional logic is inserted into the data path, not only once per register but for each combinational cell. This causes two issues:

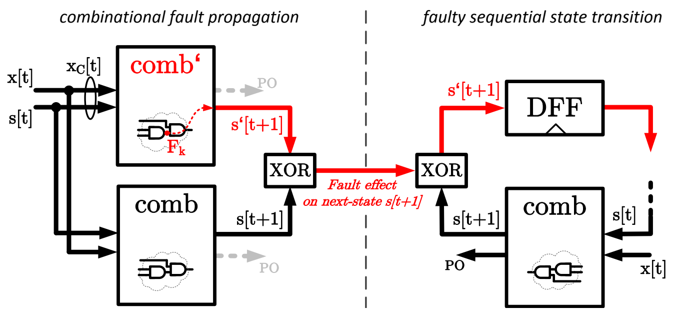
1) The circuit size of DUTs for which fault emulation can be applied is much smaller, due to limited hardware resources on the FPGA.
2) The critical timing path gets longer. We experienced a 68% reduced maximum operating frequency after having instrumented the combinational logic of an 8051-like micro-controller.

II. Proposed Method

Our goal is to overcome the issues of FPGA-based fault emulation w.r.t. combinational fault modeling. We achieve this with a new combined approach of combinational fault propagation into sequential cells, utilizing the propagate function of the SAT-solver MiniSAT [9], and a fast FPGA-based fault emulation of faults in sequential cells [2], [10]. By splitting up the problem and using a combined approach, we are able to inject transient faults into combinational and sequential cells of a DUT, whereas existing work either focuses on sequential or combinational fault injection. Additionally, it can be applied to much bigger circuits and it is faster compared to existing work, as we will demonstrate in the result section.

Fault injection in sequential cells is handled by the FPGA-based fault emulator by default and only in case of fault injection into combinational cells, we split the problem of fault modeling into two steps, as depicted in Figure 1: First, the assumption[1] function of the SAT-solver is used to inject transient faults F_k into the combinational logic *comb'* and to apply the stimuli $x_C[t]$ at combinational top-level inputs. In the very first cycle t of fault propagation, the propagate function of the SAT-solver is used to map transient faults in combinational logic to equivalent transient faults at the input of sequential cells. As depicted on the left in Figure 1, this

[1]MiniSAT's assumption function is used to assume values of Boolean variables, which is useful to constrain input variables.


combinational fault propagation
faulty sequential state transition
x[t]
x_C[t]
s[t]
comb'
PO
s'[t+1]
F_k
XOR
Fault effect on next-state s[t+1]
XOR
s'[t+1]
DFF
comb
s[t+1]
PO
s[t+1]
comb
s[t]
PO
x[t]
Figure 1: The proposed approach divides combinational fault modeling in: combinational fault propagation of fault F_k at fault injection time t (left) and sequential fault propagation, depicted for the first faulty state transition $s[t] \rightarrow s'[t+1]$ (right)

equivalent sequential fault is determined by xor-ing the fault-free next-state $s[t+1]$, generated by a fault-free instance of the combinational logic *comb*, and the faulty next-state $s'[t+1]$, generated by the faulty combinational logic *comb'*. In the second step, the determined transient fault at the inputs of sequential cells is captured and mapped onto the fault emulator by means of xor-ing it with the fault-free next-state $s[t+1]$. In order to reach the fault injection cycle t during emulation, the fault emulator performs fault-free state transitions in advance until the fault injection cycle t is reached. The fault emulator is optimized to handle single as well as multiple transient faults in sequential cells without performance loss compared to fault free test runs. This is important to note, since a lot of single transient faults in combinational cells result in multiple transient faults in sequential cells. In order to increase the performance, only distinguishable faults, which are faults that differ in timing and affected sequential cells from any other determined fault, are mapped onto the fault emulator.

The proposed method enables us to map combinational transient faults onto a fault emulator which can inject faults in sequential cells only, allowing to profit from the high performance that FPGA-based fault emulation provides. Since fault injection into combinational logic is not implemented in hardware, the instrumentation of the DUT requires less hardware resources and operates at a higher frequency compared to other FPGA-based approaches. This makes the proposed method applicable to larger DUTs and results in a higher overall performance.

III. Experimental Results and Conclusion

We demonstrate the hardware requirements and the performance of our approach with an 8051-like micro-controller design used for security applications. The DUT consists of 10,075 combinational cells and 1911 FFs and executes a functional test lasting 8090 clock cycles. The instrumented DUT, including 1911 instrumented FFs and additional controlling hardware for fault injection, fits into 16,906 logic elements (LEs) on a Cyclone IV FPGA, which is clocked with 20 MHz (F_{max} = 22 MHz). For a comparison to previous approaches, we also implemented a second state-of-the-art setup, similar to the one presented in [7], where all combinational cells are instrumented instead. The state-of-the-art setup requires 30,232 LEs, i.e. about 79% more LEs compared to the proposed method. Furthermore, we experienced, that the maximum operating frequency of the second state-of-the-art setup is limited to $F_{max} = 7\ MHz$.

We performed fault analysis in the combinational logic of this DUT, where faults are injected into all combinational cells for each clock cycle of the functional test. Analyzing about 81.5 million faults (clock cycles of the test times the number of combinational gates) with our approach took less than eleven hours (MiniSAT and emulation), whereas the state-of-the-art approach took about 26 hours. Hence, compared to the state-of-the-art approach, the proposed method executes this fault analysis for the selected test more than two times faster, although we spent additional effort for combinational fault propagation with MiniSAT. This is a result of 1) removing indistinguishable faults from fault emulation and 2) keeping the operating frequency untouched. As we already mentioned, the state-of-the-art approach requires about 79% more hardware resources on the FPGA. By saving hardware resources, our method allows to instrument much lager designs for which the state-of-the-art approach would already fail.

We conclude, that in contrast to existing work, the proposed method provides designers with a tool to perform fault analysis in sequential as well as combinational logic while it can be applied to larger designs. In addition, it increases the number of analyzed faults in given time for verification, helping to further improve safety-critical and security designs.

References

[1] M. Ebrahimi, A. Mohammadi, A. Ejlali, and S. G. Miremadi, "A fast, flexible, and easy-to-develop fpga-based fault injection technique," *Microelectronics Reliability*, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0026271414000067

[2] R. Nyberg, J. Heyszl, J. Nolles, D. Rabe, and G. Sigl, "Closing the Gap Between Speed and Configurability of Multi-Bit Fault Emulation Environments for Security and Safety-Critical Designs," in *Euromicro Symposium on Digital Systems Design*, 2014, pp. 114–121.

[3] L. Entrena, M. G. Valderas, R. F. Cardenal, M. P. Garcia, and C. L. Ongil, "Set emulation considering electrical masking effects," *IEEE Transactions on Nuclear Science*, vol. 56, no. 4, pp. 2021–2025, 2009.

[4] I. Polian, J. P. Hayes, S. M. Reddy, and B. Becker, "Modeling and mitigating transient errors in logic circuits." *IEEE Trans. Dependable Sec. Comput.*, vol. 8, pp. 537–547, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/tdsc/tdsc8.html#PolianHRB11

[5] N. Miskov-Zivanov and D. Marculescu, "Multiple transient faults in combinational and sequential circuits: A systematic approach." *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 29, no. 10, pp. 1614–1627, 2010. [Online]. Available: http://dblp.uni-trier.de/db/journals/tcad/tcad29.html#Miskov-ZivanovM10

[6] C. Lopez-Ongil, M. Garcia-Valderas, M. Portela-Garcia, and L. Entrena, "Autonomous Fault Emulation: A New FPGA-Based Acceleration System for Hardness Evaluation," *IEEE Transactions on Nuclear Science*, vol. 54, pp. 252–261, 2007.

[7] S. Kundu, M. D. Lewis, I. Polian, and B. Becker, "A soft error emulation system for logic circuits," in *Conf. on Design of Circuits and Integrated Systems*, 2005.

[8] A. Janning, J. Heyszl, F. Stumpf, and G. Sigl, "A cost-effective fpga-based fault simulation environment," *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, vol. 0, pp. 21–31, 2011.

[9] N. Eén and N. Sörensson, "The MiniSAT Page (online)," http://www.minisat.se/.

[10] R. Nyberg, J. Heyszl, D. Rabe, and G. Sigl, "Closing the gap between speed and configurability of multi-bit fault emulation environments for security and safety–critical designs," *Microprocessors and Microsystems*, May 2015.

Graph Guided Error Effect Simulation

Jo Laufenberg*, Sebastian Reiter†, Alexander Viehl†, Oliver Bringmann*, Wolfgang Rosenstiel*

* Universität Tübingen
Sand 13
D-72076 Tübingen, Germany
[laufenbe, bringman, rosenstiel]@informatik.uni-tuebingen.de

† FZI Forschungszentrum Informatik
Haid-und-Neu-Str. 10-14
D-76131 Karlsruhe, Germany
[sreiter, viehl]@fzi.de

***Abstract*—The increasing number of complex embedded systems used in safety-relevant tasks produces major challenges in the field of safety analysis. This paper presents a simulation-based safety analysis that will overcome these challenges. The presented approach consists of two parts: an Error Effect Simulation (EES) and a graph-based specification. The EES is composed of a system simulation with fault injection capability and a generic fault specification. The graph-based specification approach guides systematically the EES and enables a very efficient exploration of the analysis space. Inherent in the graph-based specification is the documentation of the safety analysis and a coverage approach to assess the executed safety analysis. Combining these parts leads to an efficient and automatable framework for safety analysis.**

I. Introduction

In the last decades the amount of software and embedded systems has increased exponentially. In the automotive domain software has become the driving factor for innovations. Current premium segment cars include more than 70 connected interacting embedded platforms. The same trend can be seen in other domains, where amount, complexity and interaction of electronic based systems are rapidly increasing. These systems are often involved in safety-relevant tasks. With this significant increased complexity the probability of operational errors is increasing too. Originated in unforeseen application scenarios, as well as fault scenarios there are many potential fault sources which may not be identified during isolated component analysis. Also the impact of external or internal interference that may occur by combining these tested components to an interconnected system may lead to new fault sources. An erroneous delivered service of these safety-relevant applications could result in disastrous accidents that may harm people's life. Therefore a reliability assessment of the overall system is an obligatory task. The complexity of the complete system and all its variability has to be handled by this reliability assessment. In the area of system verification, simulation-based assessments have been established for verification. System-level simulations are used to verify functional and non-functional requirements such as timing and power. These so called virtual prototypes (VPs), behavioral models of the system, provide a promising approach for safety assessment [1]. They characterize the interdependency between components and the explicit and inherent error tolerance of the system.

This paper presents a comprehensive approach for safety analysis based on system simulations. The work covers a modularized, configurable system simulation approach extended with fault injection capabilities. The simulation framework is designed to support both design decisions in early phase with regard to multiple system variations and detailed analysis with integrated software prototypes in later phases. An integral fault description enables the injection of diverse fault behaviors and therefore supports a variety of abstraction levels. The usability of the Error Effect Simulation (EES) is significantly enhanced by a graph-based specification approach. This approach is used to specify, control and supervise the complete safety verification as well as each single EES. It interacts with the EES before, during and after each simulation run. The graph-based specification and the provided automated exploration enables to cope with the increasing complexity introduced by the combination of multiple system alternatives, faults and system environments.

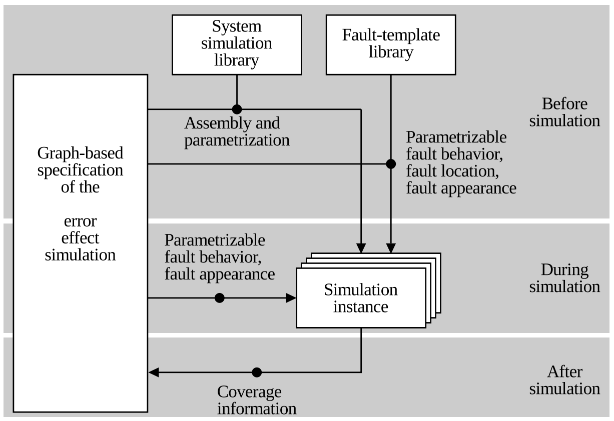

System simulation library
Fault-template library
Graph-based specification of the error effect simulation
Assembly and parametrization
Parametrizable fault behavior, fault location, fault appearance
Before simulation
Parametrizable fault behavior, fault appearance
Simulation instance
During simulation
Coverage information
After simulation

Fig. 1. Graph guided specification approach for error effect simulation

In early design phase a variety of system alternatives and parameterizations are specified with the help of a graph-based format and automatically explored by the framework. In later phases a comprehensive safety assessment is executed, with regard to manifold fault cases in different combinations and locations. The complete set of evaluations is guided by a graph-based specification, enabling an easy quantitative assessment of the overall system safety. Fig. 1 depicts the presented framework. The graph-based control module configures the EES in advance by assembling the simulation instance and parameterize the components. Additionally different fault injection behaviors are selected, the location chosen and the fault activation condition determined. During Simulation the graph-based control module directs the simulation, with focus on the injected faults. After simulation, coverage information is annotated in the graph-based specification format.

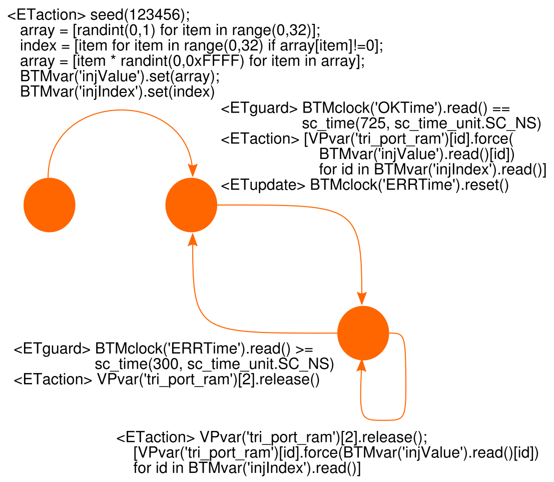
The remainder of this paper is structured as follows. After an overview of existing work for fault injection and verification control the Section III introduces the used EES framework with a generic fault specification approach. Section IV shows the graph-based specification of a verification plan. A close integration of the graph-based specification within the EES, enabling a fine-grained control of the simulation-based safety analysis is presented in Section V.

II. Related Work

Functional verification approaches can be divided into two main groups: static and dynamic verification techniques [2]. Static verification such as formal verification does not execute the source code, while for dynamic verification approaches, such as code coverage or functional coverage, the program needs to be executed. Simulation-based techniques are part of the dynamic approaches and are the leading techniques. In the development process verification plans are used to specify which components are verified and how [3]. These plans are often formulated in natural language with a high level of abstraction, given a wide scope of interpretation. To overcome these lacks, a graph-based approach is used, which is more restrictive to the scope of interpretation and gives a higher support to automation. Industrial applications like inFact designed by MentorGraphics [4] or Trek from Breker [5] use such graphical descriptions to derive test cases.
Simulation fault injection (SFI) as part of the dynamic verification is an important and established approach for safety analysis. The group of non-invasive approaches [6], [7] use compiler or simulator modifications to inject faults. Our approach should be simulator and compiler independent to enable the usage of third party tools such as [8], [9]. Another category of approaches use the so called saboteur approach. They modify the structure of the simulation and introduce additional modules that inject faults in the data path. Different forms of this approach are presented in [10], [11], [12]. Although the change within the simulation structure is best suited with our configuration approach, we did not use this approach because of its limitation to inject faults only in the exchanged data. Functional, abstract simulations often provide a monolithic structure that is not suited with the saboteur approach. Additionally the non-functional properties are most likely specified within the internal state of modules and not at the exchanged data. Therefore we chose the mutation approach. The references [13], [14], [15] present different mutation approaches. Especially the approach in [14] is similar to the presented injection technique. The main difference is in the specification of the fault behavior. None of the mentioned approaches has a comprehensive fault specification approach. Most approaches use a fixed injection behavior in the injector, a predefined set of fault behaviors with simple trigger conditions such as time dependent fault triggers. A comprehensive fault specification approach, where both the injected value and the fault trigger depend on the current state of the system simulation is not presented.
This shortcomings lead to the development of our own injection and simulation framework, used in this paper. Combining this framework with the graph-based analysis specification provides a holistic approach for safety analysis that is not targeted by previous works.

III. Error Effect Simulation

EES enables the execution of what-if analyses in presence of faults and safety mechanisms. One focus of the presented framework is the applicability across the design process and for various levels of abstraction. When using the EES in early design phases the evaluation of different design decisions is one very important goal. Using the approach in late phase the integration of SW-prototypes and a detailed evaluation of the different safety concepts have to be supported. To reduce the overhead, the EES has to reuse already specified system simulations as much as possible. The system simulation consists


```
<ETaction> seed(123456);
  array = [randint(0,1) for item in range(0,32)];
  index = [item for item in range(0,32) if array[item]!=0];
  array = [item * randint(0,0xFFFF) for item in array];
  BTMvar('injValue').set(array);
  BTMvar('injIndex').set(index)

<ETguard> BTMclock('OKTime').read() ==
              sc_time(725, sc_time_unit.SC_NS)
<ETaction> [VPvar('tri_port_ram')[id].force(
                BTMvar('injValue').read()[id])
                for id in BTMvar('injIndex').read()]
<ETupdate> BTMclock('ERRTime').reset()

<ETguard> BTMclock('ERRTime').read() >=
              sc_time(300, sc_time_unit.SC_NS)
<ETaction> VPvar('tri_port_ram')[2].release()

<ETaction> VPvar('tri_port_ram')[2].release();
   [VPvar('tri_port_ram')[id].force(BTMvar('injValue').read()[id])
   for id in BTMvar('injIndex').read()]
```

Fig. 2. Example of a Behavioral Threat Model (BTM)

of a library of parameterizable modules that are interconnected to create a simulation instance [16]. By using these basic building blocks, it is possible to create different system configurations by just changing the assembly, e.g. to exchange the used communication bus. Based on the parameterization of the basic building blocks it is possible to extend the system exploration space by changing the module parameters, such as the clock frequency of a microcontroller. The assembly and the parametrization are applied at simulation runtime by an IP-XACT configuration file. This enables the simulation of multiple system configurations without the need of re-compilation, reducing the analysis effort.
Besides the modular, configurable structure of the system simulation, EES provides an infrastructure to inject faults within the system simulation. The fault injection infrastructure provides injection and monitoring probes that are added to the system simulation. The injection probes provide an interface to change the value of the associated simulation variable, by an injection control module. The injection probes are added to the system simulation by replacing data types with an injectable probe data type. This requires a modification of existing simulation models. By using a template based data type structure that reflects the calling conventions of the reference counting `shared_ptr` of C++11, the changes are mostly limited to replacing the variable declaration. Behavioral modifications of the simulation modules are not required. To minimize the manual user modifications, especially for abstract simulation models, different extensions are provided by the injector probe, such as the possibility to specify a system simulation specific re-evaluation. Injecting a fault affects the surrounding components. With simulation directives in low level models, such as `sc_signal` this is handled by the simulation kernel, because events signaling a changed value trigger process executions. With abstract functional models or Transaction Level Modeling (TLM) simulations, this is often not the case. In this case the user has to specify which functions have to be called or which events have to be notified. In the presented framework the user can externally specify a re-evaluation strategy with full access to the affected module. This approach was chosen, to prevent the re-writing of the model with sensitive processes. Another aspect is to support diverse simulation models different restore strategies.
Errors can occur and restored in various ways, especially when supporting a wide range of abstraction levels. The injection framework has to support this various restore strategies. Injecting the impact of a paper jam in a model of printing machine control, which is fixed by a service technician, requires

restoring the system state at the time of injection. Corrupting memory cells in RTL models, the transient error is recovered when a subsequent write overwrites the erroneous value. In case gate level simulation and a short-to-ground the last driven signal, during injection, has to be restored.
While the injection probes enable a write access to the system simulation, monitoring probes enable read access to the system simulation. Because both public and private variables can be replaced, both the internal state of modules and the exchanged data or signal values can be targeted by the injection. This enables the altering of both functional and non-functional aspects of the system simulation, such as timing aspects that are often in internal variables. The probes are not associated with any injection behavior; they only provide an interface for the injection control module. This module reads an interpretable fault specification during runtime, using the probes to inject the faults. This extends the configuration file approach and enables the simulation of different injection behaviors and faults without re-compilation. The specification format is called Behavioral Threat Model (BTM) and is based on Timed Automata (TA). It is a Mealy machine that consists of a finite set of locations L, actions Σ and a set of clocks C which can be reset individually. An edge is a transition from location l_i to location l_j with an action $\sigma \in \Sigma$, a guard g and clock reset $r(C)$. Actions use the injector probes to inject faults within the simulation. Similar to a classic TA the guards depend on the local clocks. In the context of the BTMs the guards additionally depend on the current simulation state, accessed via monitoring probes. To synchronize multiple BTMs local events S are provided to synchronize transitions of different BTMs. Besides the local, resettable clocks the BTM provides local variables, accessed via the same probe interfaces to store intermediate information. For example a local counter that keeps track of the injected variables. The actions and guards are specified using the Python language. The injection control module provides a Python interpreter to evaluate guard statements and to execute actions. Therefor actions and guards can use all the functionality of Python, e.g. the pseudo random number generators. Fig. 2 highlights an example BTM that injects randomly distributed, transient faults into an array. In this case it injects faults into a register set of 32 registers. It uses BTM local variables to store the injection mask and Python expressions to handle the arrays. The injection triggering is time dependent in this example. Using a state-based specification format that is often used to model complex real-time systems offers different advantages. It is best suited to describe complex fault behavior. With an additional edge it is easily to differentiate between permanent and transient faults. With the suitable guard statements intermittent faults can be specified. Especially with abstract models, the fault behavior can be quit complex. Another advantage is that it is used to bridge the gap between low-level fault models and abstract system fault models. Well-known, low-level fault models are targeting system parts that are abstracted in system-level models. The neglected system parts would propagate this fault to a higher system-level, e.g. the communication controller that propagates noise on the channel to complex transmission errors such as data corruption or transmission delay. To enable the injection of such faults into abstract models this propagation has to be specified within the fault description. E.g. to model protocol behavior in the BTM and use original bit error probability to derive the injection probability and value. Fig. 3 shows the EES, with the required information to execute a simulation. The configuration file has to specify the assembly and parametrization of the Design Under Test (DUT),

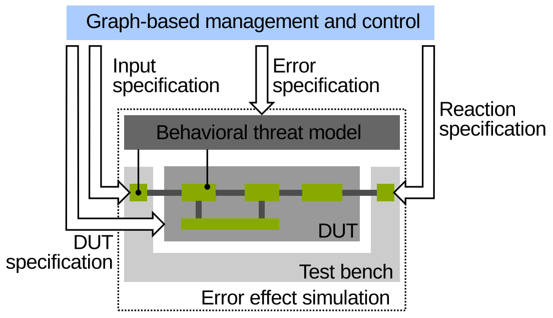


Fig. 3. Error effect simulation with graph-based control

the system input, the injection behavior that is stimulated during simulation and the expected system reactions. In the following an approach is presented that allows to generate automatically all this information.

IV. Safety Verification Plan

During EES one dedicated system instance is evaluated by applying a single injection scenario, consisting of BTMs and system stimulation. To achieve a comprehensive system analysis, different simulation runs with a variety of injection scenarios and system stimulations have to be executed. In the domain of system verification, graph-based verification plans are established to document and partially guide the verification process. In Fig. 4 a verification plan is shown. Different parts of EES are specified as nodes: the system instance (DUT) and its parametrization (DUT-C), the system input (Input), generated by the test bench, the used injection behavior (ES) and a set of checks (Check) to verify the system reaction. A path through this graph presents one system simulation. Each evaluated path creates one configuration file and executes the simulation. With the help of alternative paths a comprehensive set of analyses can be specified. The interaction between the graph and the EES is realized by the configuration file.
Brekers Trek [17] is a graph-based constraint solver developed for creating functional verification tests for digital designs and applied in this work. Trek provides two types of nodes: diamonds, which specify a selection of the subsequent nodes and rectangles, which specify an unordered sequence. The path evaluation can be executed randomly or guided via forcing and masking nodes, to fulfill e.g. coverage criteria. The evaluated paths can be visualized and the node and path coverage can be calculated automatically.

V. Simulation Control

Besides the global control of the safety analysis, Trek can be used to realize dynamic injection campaigns. The BTM's state-based specifications offers a first degree of flexibility and enables the modeling of the injection scenario with different states and a variety of injection behaviors. Specifying a whole injection campaign with a sequence of multiple injection behaviors would be possible but for each new assembly a new

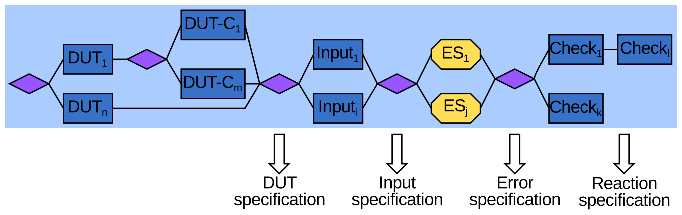


Fig. 4. Verification plan for error effect simulation

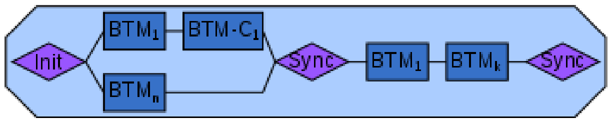


Fig. 5. Error injection plan during simulation

BTM has to be generated. Trek enables assigning basic BTMs to nodes and adding different parametrization alternatives. To get the required flexibility the graph-based specification is evaluated in parallel to the system simulation and can therefore react on events during the simulation. Synchronization between Trek and the SystemC-based simulation is realized with dedicated statements in the BTM. The BTM interpreter enables registering callback functions that can be called by guarded state transitions. This way a callback function to the Trek graph-evaluation can be registered, forwarding information from the system simulation to Trek. The Trek environment evaluates this information and calculates the next BTM. In this case different modes are possible: In the first mode, the graph is further evaluated, so only BTMs are chosen, which affect components reacting to the output of the component influenced by the actual active BTM. The second mode restarts Trek from the beginning of the system model (holding the chosen system specification and input), so every defined BTM can be calculated as next. In this case we can use constraints, two limit the possibilities. In a third mode we can choose to repeat the actual BTM with a different parametrization. The EES provides an interface for Trek to substitute the currently active BTMs. Therefore, Trek can dynamically change the injection behavior during the simulation. Fig. 5 shows an injection scenario with multiple BTMs, BTM configurations and the synchronization points between Trek and the EES. At each synchronization point the simulation is executed until the callback condition is met and Trek is activated again.

With this approach the different injection strategies can be easily assembled from basic BTMs and the overall injection campaign is well documented. Combining the verification plan generation of the previous section with the injection control presented in this section, results in the following procedure: First the DUT and its parametrization is created, then the used system input is selected. The first BTM specification and the synchronization points are created and all information is stored in the EES configuration file. The EES is executed and every time a synchronization point is reached the control is given to Trek to evaluate the current simulation state and calculate the next injection behavior. This behavior overwrites the currently active injection strategy and the EES is continued until the next synchronization point. After the EES is finished, the specified checks are asynchronously validated with the recorded trace files. Another approach, where assertions are generated due to the checks and validated during simulation, is evaluated too. In this case the assertions are added to the configuration file. After simulation of one path, Trek will calculate the remaining possible paths, each time executing an EES simulation. With this tooling environment a complete safety analysis with multiple system alternatives, different injection scenarios and system stimuli can be easily specified and automatically executed.

VI. Conclusion

In this work an approach is presented that executes a safety analysis with the help of simulation-based fault injection. A modular, reconfigurable system simulation with injection capability builds the core of the analysis and enables the easy evaluation of multiple system alternatives. A graph-based tooling environment reduces the analysis effort significantly, by automatically deriving multiple simulation runs from a graphical specification. The user specifies the atomic parts of the analysis and connects them with choice and sequence operations in a graph-based editor. The tool calculates all possible paths, each representing one system simulation. This way the complete safety analysis can be derived from one graph-based specification. Different coverage approaches process the executed simulation runs and give the user an assessment of the executed safety analyses.

Acknowledgement

This work has been partially supported by the German Ministry of Science and Education (BMBF) in the project EffektiV under grant 01IS13022.

References

[1] J.-H. Oetjens, N. Bannow, M. Becker, O. Bringmann, and B. et al., “Safety evaluation of automotive electronics using virtual prototypes: State of the art and research challenges,” in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, June 2014, pp. 1–6.

[2] K. Batcher and C. Papachristou, “Instruction randomization self test for processor cores,” in *Proceedings of the 1999 17TH IEEE VLSI Test Symposium*, ser. VTS ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 34–. [Online]. Available: http://dl.acm.org/citation.cfm?id=832299.836543

[3] B. Wile, J. Goss, and W. Roesner, *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[4] Mentor Graphics. (2015, Jan) ModelSim. [Online]. Available: http://www.mentor.com/products/fv/infact/

[5] Breker. (2015, Sep) brekersystem. [Online]. Available: http://www.brekersystems.com/products/trek/

[6] P. Lisherness and K.-T. Cheng, “SCEMIT: A SystemC error and mutation injection tool,” in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, 2010, pp. 228–233.

[7] D. Lee and J. Na, “A novel simulation fault injection method for dependability analysis,” *Design Test of Computers, IEEE*, vol. 26, no. 6, pp. 50–61, Nov 2009.

[8] cadence. (2013, Dez) Incisive Enterprise Simulator Datasheet. [Online]. Available: http://www.cadence.com/rl/Resources/datasheets/incisive_enterprise_specman.pdf

[9] Mentor Graphics. (2015, Jan) ModelSim. [Online]. Available: http://www.mentor.com/products/fv/modelsim/

[10] K.-C. Chang, Y.-C. Wang, C.-H. Hsu, K.-L. Leu, and Y.-Y. Chen, “System-bus fault injection framework in systemc design platform,” in *Secure System Integration and Reliability Improvement, 2008. SSIRI ’08. Second International Conference on*, 2008, pp. 211–212.

[11] K. Rothbart, U. Neffe, C. Steger, R. Weiss, E. Rieger, and A. Muehlberger, “High level fault injection for attack simulation in smart cards,” in *Test Symposium, 13th Asian*, Nov 2004, pp. 118–121.

[12] S. Misera, H. Vierhaus, L. Breitenfeld, and A. Sieber, “A Mixed Language Fault Simulation of VHDL and SystemC,” in *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, 2006.

[13] N. Bombieri, F. Fummi, and G. Pravadelli, “A Mutation Model for the SystemC TLM 2.0 Communication Interfaces,” in *Design, Automation and Test in Europe, 2008. DATE ’08*, March 2008, pp. 396–401.

[14] R. Shafik, P. Rosinger, and B. Al-Hashimi, “SystemC-based Minimum Intrusive Fault Injection Technique with Improved Fault Representation,” in *International On-line Test Symposium (IOLTS)*. IEEE Computer Society, April 2008.

[15] A. Fin, F. Fummi, and G. Pravadelli, “AMLETO: a multi-language environment for functional test generation,” in *Test Conference, 2001. Proceedings. International.*

[16] S. Reiter, A. Burger, A. Viehl, O. Bringmann, and W. Rosenstiel, “Virtual Prototyping Evaluation Framework for Automotive Embedded Systems Engineering,” in *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools ’14, 2014.

[17] J. Behrend, G. Dittmann, K. Keuerleber, J. Grosse, and F. Krampac, “Graph-Based Verification Patterns,” in *ACM/IEEE Design Automation Conference (DAC) – Designer Track (poster)*, ser. DAC ’13, 2013.

Towards Generating Test Suites with High Functional Coverage for Error Effect Simulation

Aljoscha Windhorst[1] Hoang M. Le[1] Daniel Große[1] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{windhorst, hle, grosse, drechsler}@uni-bremen.de

Virtual Prototyping (VP) has become an integral part of the development process for hardware-software systems in industrial settings. Implementing VPs at the *Electronic System Level* (ESL) [1] in particular can massively reduce development costs by enabling early software development and efficient design space exploration. The application of verification techniques at high abstraction levels allows for early bug detection and can thereby massively reduce the time spent for development and testing of the system. In practice, ESL models are often implemented using SystemC [2], following a behavioral/algorithmic style in combination with abstract communication based on *Transaction Level Modeling* [3]. Because of the known limitations of formal verification techniques with respect to large real world designs, simulation based verification is still the method of choice to ensure correctness of these models. However, the significance of verification results for the overall quality of the resulting product highly depends on the quality of the test suite (or alternatively, the verification scenarios) in use. Therefore, a range of coverage metrics have found their way into industrial verification flows [4]. They can be categorized roughly as either *code coverage* or *functional coverage* metrics, the former determining the subset of code executed, the latter determining the subset of system functionality triggered by the corresponding input sequences.

Multiple approaches have been proposed to both increase coverage and reduce time necessary for directed testing using different notions of *random stimuli generation*. In contrast to the naive approach of merely selecting input stimuli at random, hoping to reach system states not taken into account by the verification engineer, *Constrained Random Verification* (CRV) tries to increase the chance of selecting meaningful values in the first place. This is achieved by constraining the stimuli generation using a set of user defined properties derived from the system specification [5], [6], [7], thus “directing” the stimuli generator towards potentially interesting scenarios. To further increase the efficiency of stimuli generation, techniques such as mutation analysis, Markov chains or Monte Carlo methods have been applied to CRV [8], [9].

In resiliency evaluation, *Error Effect Simulation* is currently an active field of research, in particular at early design phases using VPs [10]. Error effect simulation essentially allows a what-if analysis of the VP, assuming that errors are present. However, only effects resulting in behavior that has been covered by the verification scenarios can be exposed. Therefore, a high quality suite of scenarios addressing potential system faults is of utmost importance. Unfortunately, the ability of resilient systems to detect and counteract these faults naturally extends the system’s state space and thereby poses an increased challenge to the stimuli generator.

In this paper we propose a new technique for automated generation of input sequences to further increase functional coverage of verification scenarios while keeping the time spent on verification low. It incorporates cover group modeling [4] as our means of directing state exploration. Selective symbolic execution of the *Device Under Verification* (DUV) is used for actual state space traversal. Practically, this is accomplished by executing the unmodified SystemC model and a specific testbench using the S^2E platform [11] extended by our own custom plugin for functional coverage support. While existing frameworks usually rely on custom SystemC frontends or intermediate representations supporting only a subset of SystemC and C++ [12], [13], S^2E performs symbolic execution on machine code level. Therefore, no functional limitations apply to the implementation of the model.

In the following, we describe the main concepts of our approach. Before this, we briefly clarify the main notions in the context of functional coverage [14], specifically cover group modeling. A cover group specifies a set of signals, called cover points, at a certain time in simulation. For each cover point, a set of cover bins is defined to represent a set of values or value ranges. Whenever the value sampled at a cover point lies within the set of a cover bin, it is considered hit. The number of hits per bin is counted during simulation and, in the end, compared to the coverage goal specifying a minimal number of hits for each bin. If this minimum is reached for every cover bin, the system functionality is considered sufficiently covered by the verification scenarios.

Since we are able to perform a symbolic execution, we can make use of the cover groups as follows. The execution engine tries to explore every feasible path of the DUV, beginning with the test bench, as shown in the exemplary *Control Flow Graph* (CFG) in Fig. 1. Every time a conditional statement is reached, the execution continues, for the time being, in either the then or else branch, adding the appropriate condition to the *path condition*, representing all choices down to the current execution state. Once the desirable execution depth has been reached, an input sequence can be generated by assigning all variables with values that satisfy the path condition. The execution then continues by backtracking to an already executed conditional statement and following the opposite branch until every branch has been visited.

While this already guarantees a complete coverage of all reachable branches, important system states, such as error

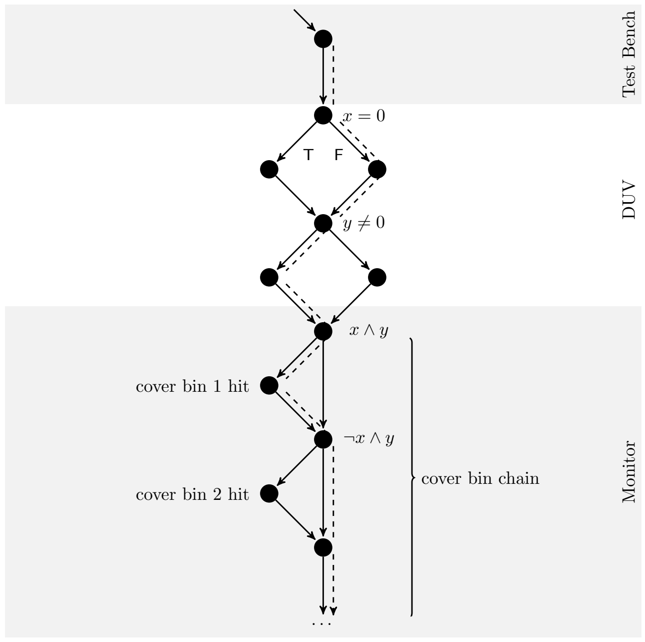


Fig. 1. Exemplary CFG with cover bin chain

states, could be missed. This is visualized in the path represented by the dashed line in Fig. 1. It is clearly possible to achieve full branch coverage inside the DUV without following this particular path, meaning there will be no scenario comprising the condition $x \wedge y$. This is where cover bins come into play. In our implementation, a cover bin is technically an if-statement comprising the specified value as its condition. The symbolic execution will eventually reach the monitor containing the *cover bin chain*, i.e. a chain of conditional statements inquiring the state of every cover bin. The execution engine will try to execute every branch of the chain, resulting in scenarios considering every system state specified by the cover bin definition. Clearly, to visit the then branch of cover bin 1 in Fig. 1, the execution engine has to follow the indicated path first.

Unlike CRV constraints, cover points can be defined on input signals as well as on internal or output signals. This allows to notably simplify the task of filling coverage gaps: Instead of reverse engineering the system design to determine input constraints that trigger a certain event potentially deep inside the system, the execution engine will automatically determine an execution path to the specified system state, including the necessary input values, if any such path exists. This is especially useful for error effect simulation, because the injected errors frequently affect internal signals and the determination of matching inputs is non-trivial.

With the proposed technique we cannot only generate verification scenarios, but also identify cover bins which cannot be hit. These could be the result of bugs either in the DUV or in the cover bin definition. Further evaluation of these cases could provide valuable information for debugging.

First experiments have been performed to evaluate the feasibility of the proposed method and to identify possible challenges. Small VPs ($<$1000 loc) with a limited number of states and cover bins were completely executed within minutes. We always reached full coverage, which was not achieved by random stimuli generation.

REFERENCES

[1] G. Martin, B. Bailey, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann, Jul. 2010.

[2] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.

[3] L. Cai and D. Gajski, "Transaction Level Modeling: An Overview," in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '03. New York, NY, USA: ACM, 2003, pp. 19–24.

[4] G. Allan, G. Chidolue, T. Ellis, H. Foster, M. Horn, P. James, and M. Peryer, "Coverage Cookbook," 2012. [Online]. Available: https://verificationacademy.com/cookbook/coverage

[5] J. Yuan, C. Pixley, and A. Aziz, *Constraint-based Verification*. Springer, 2006.

[6] F. Haedicke, H. Le, D. Grosse, and R. Drechsler, "CRAVE: An advanced constrained random verification environment for SystemC," in *2012 International Symposium on System on Chip (SoC)*, Oct. 2012, pp. 1–7.

[7] M. F. S. Oliveira, C. Kuznik, W. Mueller, F. Haedicke, H. M. Le, D. Große, R. Drechsler, W. Ecker, and V. Esen, "The system verification methodology for advanced TLM verification," in *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2012, pp. 313–322.

[8] T. Xie, W. Mueller, and F. Letombe, "Efficient Mutation-Analysis Coverage for Constrained Random Verification," in *Distributed, Parallel and Biologically Inspired Systems*, ser. IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg, 2010, no. 329, pp. 114–124.

[9] N. Kitchen and A. Kuehlmann, "Stimulus Generation for Constrained Random Simulation," in *Proceedings of the 2007 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '07. Piscataway, NJ, USA: IEEE Press, 2007, pp. 258–265.

[10] J.-H. Oetjens, N. Bannow, M. Becker, O. Bringmann, A. Burger, M. Chaari, S. Chakraborty, R. Drechsler, W. Ecker, K. Grüttner, T. Kruse, C. Kuznik, H. M. Le, A. Mauderer, W. Müller, D. Müller-Gritschneder, F. Poppen, H. Post, S. Reiter, W. Rosenstiel, S. Roth, U. Schlichtmann, A. von Schwerin, B.-A. Tabacaru, and A. Viehl, "Safety evaluation of automotive electronics using virtual prototypes: State of the art and research challenges," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 113:1–113:6.

[11] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A Platform for In-vivo Multi-path Analysis of Software Systems," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 265–278.

[12] K. Marquet and M. Moy, "PinaVM: A SystemC Front-End Based on an Executable Intermediate Representation," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, ser. EMSOFT '10. New York, NY, USA: ACM, 2010, pp. 79–88.

[13] H. Le, D. Grosse, V. Herdt, and R. Drechsler, "Verifying SystemC using an intermediate verification language and symbolic simulation," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, May 2013, pp. 1–6.

[14] "IEEE standard for SystemVerilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)*, pp. 1–1315, Feb 2013.

Accurate Cache Vulnerability Modeling in Presence of Protection Techniques

Yohan Ko*, Reiley Jeyapaul**, Youngbin Kim*,
Kyoungwoo Lee*, and Aviral Shrivastava***
*Department of Computer Science, Yonsei University, Seoul, Korea
**ARM Research, TX, USA
***Department of Computer Science and Engineering, Arizona State University, AZ, USA
Email: *{yohan.ko, yb.kim, kyoungwoo.lee}@yonsei.ac.kr
**reiley.jeyapaul@arm.com ***aviral.shrivastava@asu.edu

Cache is one of the most susceptible components to failures due to increasing soft errors [1] in a processor. Several protection techniques have been proposed to improve the cache reliability. ECC (Error Correction Code) and parity protections are the most common ways due to their design simplicity and effectiveness. However, cache protection techniques incur significant overheads in terms of power, area, and performance [2]. There is no accurate method to represent the effectiveness of applied protection techniques in processors. Thus, it is a necessity to present the accurate cache reliability in the presence and absence of protection techniques.

Fault injection can evaluate the effectiveness of protection schemes against soft errors [3]. Faults are injected into a specific bit of microarchitectural components (e.g., cache memory) at the specific timing during the execution, and we can decide whether it induces the system failure or not. Fault injection is one of the most accurate schemes to measure the failure rate of microarchitectural components. However, fault injection is hard to correctly set up, and it also takes lots of execution time and computing resources to run thousands of fault injections.

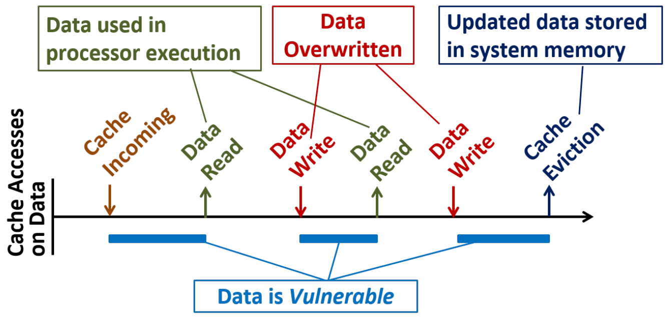


Fig. 1. Vulnerability estimation scenario of write-back cache. Read and eviction at the dirty state make the vulnerable period since soft errors at that time can be propagated to other components, e.g., CPU or lower-level memory.

Vulnerability is an alternative metric to estimate the susceptibility of data in microarchitectural components [4]. It is important to note that not all hardware bits in a cache memory are susceptible to soft errors during all the execution time. For example, if a bit in the cache is overwritten by the write operation before being used or read, then even if a soft error happens on it, it will not be vulnerable as shown in Fig. 1. On the other hand, the read operation and the eviction at the dirty state make the period vulnerable since soft errors at that time can be propagated to the other components such as CPU or lower-level memory. Cache vulnerability is estimated in bits $\times$ cycles in which bits are vulnerable, and we have performed vulnerability analyses for each cache block for the entire program to estimate the vulnerability of the entire cache.

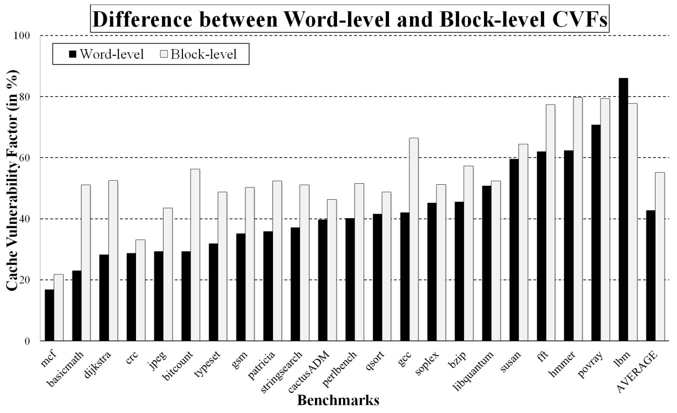


Fig. 2. Inaccuracy of block-level CVF estimations (vs. word-level)

We have presented gemV-cache [5], the accurate and protection-aware cache vulnerability estimation toolset based on the cycle-accurate gem5 simulator [6]. First off, existing implementations that estimate the cache vulnerability at the block-level [7] can be inaccurate by 7% on average and up to 24% inaccurate as compared to word-level vulnerability estimation in our gemV-cache as shown in Fig. 2. Estimating the vulnerability for the entire cache hides the extent of errors in block-level estimation technique. This is because, for some blocks the block-level vulnerability is smaller than word-level estimation, and for others it may be larger. The sum of absolute vulnerability difference of each block, the sum of vulnerability underestimation and overestimation, can be inaccurate on average by 34%.

Inaccurate vulnerability estimation based on block-level behaviors can have negative impact on the implementation of protection techniques and their effectiveness. For example, suppose that we want to protect only 3 cache blocks by exploiting ECC protection in a benchmark *basicmath*. Then, we should estimate the vulnerability of each cache block, and

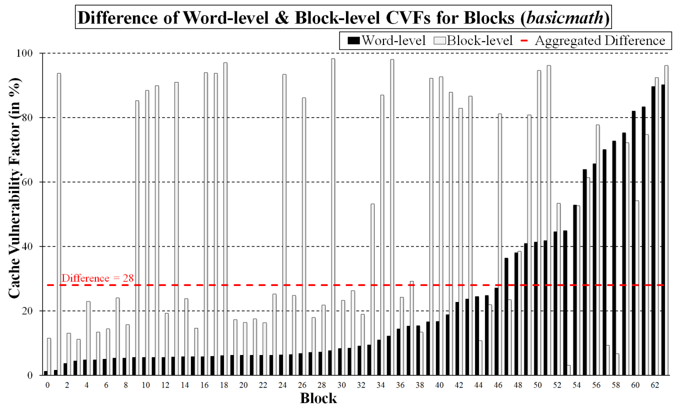


Fig. 3. Dramatic difference of block-level and word-level CVF for each block

then protect the 3 blocks with higher vulnerability than others. If we make this analysis and select 3 blocks based on the block-level estimation, we would expect 22% vulnerability reduction as shown in Fig. 3. However, the accurate analysis based on our word-level estimation shows that the selected blocks are not actually the most vulnerable ones, and the inaccurate selection would actually result in only 3% reduction in vulnerability.

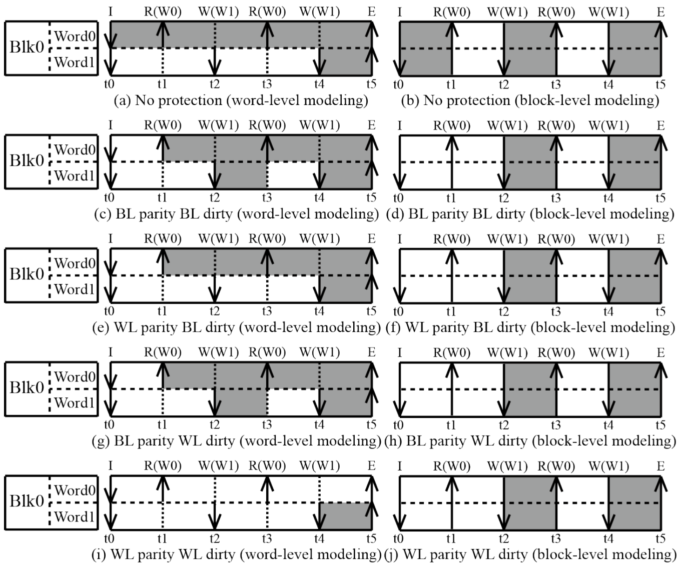

I R(W0) W(W1) R(W0) W(W1) E
Blk0 Word0 Word1
t0 t1 t2 t3 t4 t5
(a) No protection (word-level modeling)
(b) No protection (block-level modeling)
(c) BL parity BL dirty (word-level modeling)
(d) BL parity BL dirty (block-level modeling)
(e) WL parity BL dirty (word-level modeling)
(f) WL parity BL dirty (block-level modeling)
(g) BL parity WL dirty (word-level modeling)
(h) BL parity WL dirty (block-level modeling)
(i) WL parity WL dirty (word-level modeling)
(j) WL parity WL dirty (block-level modeling)

Fig. 4. Word- and block-level modelings with parity under the same scenario (*shaded region: vulnerable periods, BL: Block-Level, WL: Word-Level*)

Secondly, our gemV-cache estimates the cache vulnerability at the word-level granularity in order to apply the granularity of protections such as parity and status bits. Assume that a word in the block has one parity bit, is this word-level parity protection more reliable than block-level one? How can the granularity of dirty bits affect the cache vulnerability? With block-level vulnerability estimation, there is no difference among the granularity of parity and dirty bits as shown in Fig. 4 since block-level estimation cannot track behaviors of each word in the same cache block separately. Thus, we should accurately estimate the vulnerability at the word-level to implement and apply protection techniques in processors.

However, it is much more involved to estimate the vulnerability at the word-level than at the block-level. The main source of complexity is the fact that as opposed to estimating the vulnerability at the block-level, when estimating the vulnerability at the word-level, the vulnerability of a word may not be independent of the accesses to the other words in the same block. For example, block-level parity protection can be more vulnerable than no protection due to behaviors of other words in the same cache block as shown in Fig. 4.

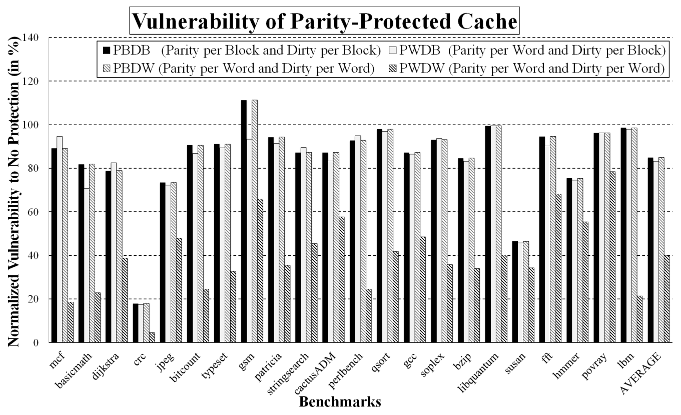


Fig. 5. Normalized vulnerability to no protection of parity-protected cache with diverse granularity of parity and dirty bits

Finally, using accurate vulnerability analysis, we explore the design of parity-protected caches. Our analysis reveals several interesting results as shown in Fig. 5.

- Parity protection can be reliable protection technique for some benchmarks such as *crc* and *susan*, but it can reduce the vulnerability by only 15% on average as compared to no protection.
- Block-level parity protection can be more vulnerable than no protection for benchmark *gsm*. It is because that read operation to a word at the dirty state makes the entire block vulnerable even though data in the other word is overwritten as described in Fig. 4 (c).
- If word-level parity protection has block-level dirty bit, it can reduce the vulnerability by only 2% as compared to block-level parity protection with block-level dirty bit. For better protection, both parity and dirty bits should be designed at the finest word-level.

References

[1] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim, "Robust system design with built-in soft-error resilience," *IEEE Computer*, 2005.

[2] N. Sadler and D. Sorin, "Choosing an error protection scheme for a microprocessor's L1 data cache," in *ICCD*, 2006.

[3] G. Asadi, S. Miremadi, H. Zarandi, and A. Ejlali, "Fault injection into SRAM-based FPGAs for the analysis of SEU effects," in *FPT*, 2003.

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

[5] Y. Ko, R. Jeyapaul, Y. Kim, K. Lee, and A. Shrivastava, "Guidelines to design parity protected write-back L1 data cache," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 24.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, 2011.

[7] W. Zhang, "Computing cache vulnerability to transient errors and its implication," in *DFT*, 2005.

On the Correlation of HW Faults and SW Errors*

Wolfgang Mueller, Liang Wu, Christoph Scheytt
Heinz Nixdorf Institut, Paderborn
Germany

Markus Becker, Sven Schönberg
C-LAB, Paderborn
Germany

Abstract—Electronic systems, like they are embedded in road vehicles, have to be compliant to functional safety standards like ISO 26262 [3]. Those standards define different safety levels, which require different means and measures for safety verification and risk analysis like fault effect simulation. In this context it is important to investigate the impact of physical and hardware related effects to higher abstraction levels. This article gives first an overview of the basic principles of fault simulation and mutation based analysis. Thereafter, it studies an example for code and model mutations through different abstraction levels. This is a contribution to clarify vertical fault relations and their impact throughout the system development process and the impact of physical effects to high abstraction levels.

I. Motivation

With the advent of advanced embedded microelectronics, we can find computers embedded almost everywhere, so that utmost care has to be taken to avoid any risk to human health or life in their operation. Therefore, each application domain has its individual standard to define functional safety requirements and regulations for their implementation and test like IEC60730-Annex H (household machines) [1], DO-254 (airborne systems) [2], ISO 26262-Part 5 (road vehicles) [3], and ISO 13849-1 (machinery) [4], which may come with additional regulations and memoranda, like EASA CM-SWCEH-001 [7]. Those standards comprise means and measures to assess the residual risk of unavoidable faults and potential hazards and to reduce them to the required safety level. As such those standards outline different categories of specific conditions, tests, and safety levels a certified system has to fulfill. ISO 26262, for instance, defines ASIL (Automotive Safety Integration Level) A-D, which introduces the risk of system failures with their impact on different levels of injuries. Along those levels, it also requires different methods to verify the robustness and operation under external stresses like EMC (Electro Magnetic Compatibility) and ESD (Electro Static Discharge), statistical test, worst-case test, over limit test, and environmental testing.

In an early step, the design of a safe and reliable system typically takes the combined application of analytical methods like Fault Tree Analysis (FTA) [5] and Failure Mode and Effects Analysis (FMEA) [6] to quantify the residual risks in terms of a safety metrics according to a safety integrity/performance level. Later steps extend to the analysis by simulation like functional and stochastic simulation once a model and implementation is available. In this context, the reliability of operations is typically measured by Mean Time between Failures (MTBF), for instance, where typical numbers for higher reliability are between 100 and 1000 years [7,8]. It is essential here to verify the fault tolerance resp. fault resistance of a system to either continue operation (fail-operate) or at least to enter a safe state (fail-safe). A fault analysis typically investigates how errors propagate to failures in terms of fault injection scenarios, which activate errors. An analysis typically requires the availability of a fault/failure statistics gathered from the system in operation or from worst-case assumptions or estimates.

Faults and errors in electronic systems can be due to design bugs, manufacturing bugs, and various lifetime effects like radiation, aging, and vibration. Here, we mainly distinguish between permanent errors (hard errors) and transient errors (soft errors), which both relate to well-known effects in low- and high-voltage semiconductors. Hard errors can be due to design, manufacturing and lifetime errors, like catastrophic Single Event Latch-ups (SEL), Burnouts (SEBO), Gate Ruptures (SEGR), or electro-migration, for instance. Soft errors can be due to strike of ionizing particles with Single Event Upsets (SEUs), power fluctuations, or electromagnetic interferences, for instance. They often manifest themselves in bit flips, which may corrupt the operating system, the main data, or the control program of a computer based system [8].

In general, system failures can be significantly reduced by various error prevention, detection, correction and tolerance techniques. Multiple general countermeasures can be implemented in hardware and software at different levels of abstraction. Examples are CRC (Cyclic Redundancy Check), ARQ (Automatic Repeat Request), watchdogs, one/multi bit redundancy, multi-channel diversity, and different types of block replications with optional path comparison and voting. To prove the effectiveness and impact of those countermeasures, fault effect simulation can be applied for the simulation of fault effects injected by the environment to verify that a system under design is resistant and/or tolerant against a set of injected faults determined by a specific fault model.

The remainder of this article first introduces the main principles of fault effect simulation and mutation-based analysis. Thereafter, it discusses the correlation and relevance of code and model mutations for fault injection between different abstraction levels before we finally close with a summary and conclusion.

II. Mutation Based Analysis

The principles of fault simulation stem from classical hardware design, where they were introduced to develop test vectors with a high fault coverage for post manufacturing circuit tests. As such, a set of test vectors (i.e., testbench) is applied to a potentially correct (i.e., golden) model and to a fault injected (i.e., mutated) model. A test vector is considered to detect an injected fault when the outputs of the

* The work described herein is funded by the Bundesministerium für Bildung und Forschung (BMBF) through the EffektiV Project (No. 01IS13022)

golden model differs from the outputs of the mutated model when applying the test vector. Note here, that a test vector can detect more than one fault. If all possible faults based on a specific fault model are detected by a set of test vectors, the set is considered to have a 100% fault coverage. This metrics finally determines the overall quality of a testbench and is thus denoted as testbench qualification. However, the final goal is always maximize the fault coverage and to minimize the number of test vectors in order to reduce the number of simulation and test runs. Therefore, it is important to identify and remove redundant test vectors, i.e., test vectors with the same and with no effect at the output.

As the fault coverage is always based on a specific fault model, the fault model is always a compromise between acceptable simulation runtimes for high coverage ratios and a realistic correlation to physical artifacts in order to detect as many faulty circuits as possible in the final test. As such, a fault model is always an abstraction of physical effects and due to a specific abstraction level. One of the first fault models was the structural stuck-at-1/0 fault model at gate level and the stuck-at-open/close fault model at transistor level [11]. Though the classical terminology of fault models and fault simulation comes from the hardware domain, we can find comparable concepts in the area of software testing, i.e., automatic fault injection into software code, which is widely known as mutation based software testing or analysis [10]. Later, with the advent of electronic system level hardware description languages, mutation based testing technologies became also accepted for hardware verification and tools became available for the testbench qualification at register transfer level, like Certitude [12]. As we apply this technology in the next section, we first give a brief overview of its basic principles before we proceed to our study.

A mutation is defined as a single fault injected into the copy of the original software code denoted as a mutant. A mutation example is given by the following C code, which takes a C statement and applies a so-called mutation operator: 'a = b + c;' ➔'a = b - c;'. Mutation operators inject faults by replacing symbols or objects in the model or source code, e.g., replacing [`+'➔`-'] or [`>'➔`≥']. By deploying pre-defined mutation operators at different locations in a model or in the source code, we can obtain a huge number of mutants. As a complete mutation analysis basically requires one simulation run for each mutant and each test vector, there are techniques to significantly reduce the number of simulation runs [12].

So far, mutation-based analysis is applied and investigated at a specific abstraction level for testbench qualification: RTL hardware models (e.g. VHDL, Verilog), software models (e.g., UML), software source code (e.g., C), and software binaries (e.g., ARMv4). However, in the context of functional safety certification, it not only important to investigate fault propagation horizontally at a specific level rather than to consider cross-level fault correlations in order to determine the real impact of physical effects, like SEUs, to the individual functions of a system. Consider, for instance, an SEU related bit flip in a memory cell, which changes the binary encoding of a software instruction [8,9]. This can also be clearly determined as a mutation of the software, so that we can identify relations to equivalent source code mutations.

The next section will investigate the design process in more details with respect to fault injection, i.e., mutations, and their correlations between different abstraction levels.

III. FAULT CORRELATIONS

Consider again the strike of a neutron or alpha particle at a semiconductor and an SEU, which results in a transient bit flip. A bit flip at a specific memory location may give the mutation of a software instruction where it finally changes a bit in the binary code of a software instruction. That bit can change the opcode, address, or data section of that instruction, which may directly correlate to a mutation in the software source code. This shows that design and runtime errors can be tightly correlated. However, at the end it is not relevant for a test if the wrong operation is introduced by a faulty memory cell or by a coding error of a programmer. When we exactly know the correlation between all levels of hardware and software faults, we can also apply the principles of software mutations for fault effect analysis to cover correlated hardware faults without changing the hardware itself. This refers to the assumption of Software-Implemented Hardware Fault Tolerance (SIHFT) and the application of very fast virtual environments like Xemu [13], which applies fault injection at binary software level.

Current practice in risk analysis assumes bit flip errors mostly on the basis of a uniformly distributed probability of bit patterns. However, to head for a more advanced risk analysis we now study the correlation of faults and their propagated errors and failures across different domains and different abstraction resp. refinement levels by a simple example with focus on a operation. Hereby, we first follow a top down approach and start with a software specification and stepwisely proceed to a software model and implementation, which is later compiled to a binary code.

$$f(a,b,c) = \begin{cases} \mathit{true}, & \overset{a > b > c}{\cancel{a \le b \le c}} \\ \mathit{false}, & \text{otherwise} \end{cases}$$

Figure 1. Software specification example.

Figure 1 shows the simple specification example of a comparison, which can be part of a more complex system specification. The given function is defined in Boolean space and specifies the relation of three arguments, which can be of any ordered types. The function is true for any abc-tuple in ascending order and false, otherwise.

Let us now apply ['≤'➔'>'] as a mutant operator, which replaces all '≤'s by '>'s in the specification. The result defines a,b,c in descending order as indicated in red in Figure 1. From that specification, we can create different software models like Matlab/Simulink or UML or as given in Figure 2 and 3.

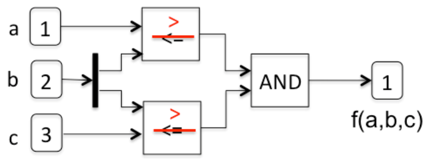


Figure 3. Matlab/Simulink model.

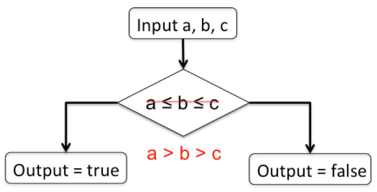
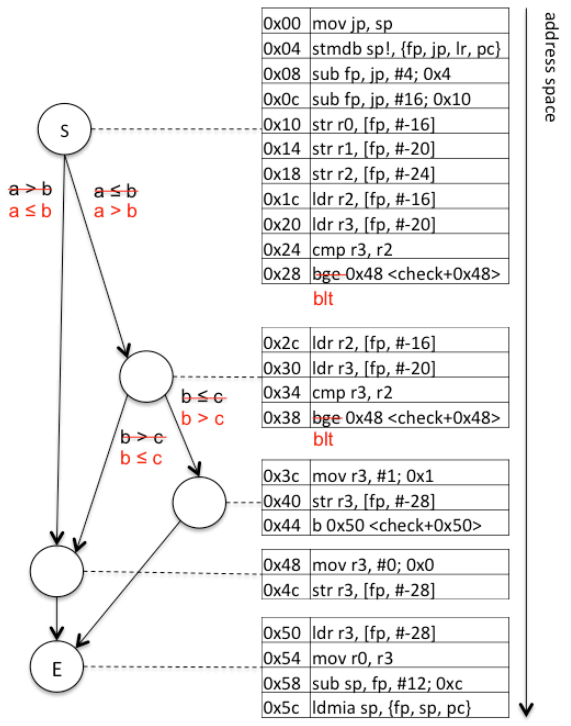


Figure 2. UML activity diagram model.

For our simple example, this step is a straightforward transformation of the specification and the given mutation operator, which has to be transferred to either control flow- or data flow-oriented means. It is important to note here that the detailed Model of Computation (MoC) of the chosen target modeling language is of major importance. Note also that our example focuses on an operator/opcode oriented approach. Data oriented approaches may apply other techniques for fault injection like reordering of inputs. However, we can identify more or less strong relationships between both approaches for several cases. The reordering mutation of inputs from a,b,c to c,b,a in Matlab/Simulink can be, for instance, equivalent to the application of [‘<’➔‘>’].



```
0x00 mov jp, sp
0x04 stmdb sp!, {fp, jp, lr, pc}
0x08 sub fp, jp, #4; 0x4
0x0c sub fp, jp, #16; 0x10
0x10 str r0, [fp, #-16]
0x14 str r1, [fp, #-20]
0x18 str r2, [fp, #-24]
0x1c ldr r2, [fp, #-16]
0x20 ldr r3, [fp, #-20]
0x24 cmp r3, r2
0x28 ~~bge~~ 0x48 <check+0x48>
     blt

0x2c ldr r2, [fp, #-16]
0x30 ldr r3, [fp, #-20]
0x34 cmp r3, r2
0x38 ~~bge~~ 0x48 <check+0x48>
     blt

0x3c mov r3, #1; 0x1
0x40 str r3, [fp, #-28]
0x44 b 0x50 <check+0x50>

0x48 mov r3, #0; 0x0
0x4c str r3, [fp, #-28]

0x50 ldr r3, [fp, #-28]
0x54 mov r0, r3
0x58 sub sp, fp, #12; 0xc
0x5c ldmia sp, {fp, sp, pc}
```

Figure 4. ARMv4 instruction level.

The refinement from the model to the software source code is typically executed by the application of a target-specific code generator. For this, the generator maps data and control flows to variables and data structures for which data types are generated. Computational operator statements must be defined accordingly. The control flow is generally mapped to programming language constructs, such as for/while loops, switch/case or if statements to which fault injection/mutation operators are applied. The source code is thereafter compiled by a target-specific compiler to a target instruction set architecture, like ARMv4 or Tricore™. Final instructions are typically composed of an opcode with optional address and data sections. As such, data flows need to be mapped to registers and memory of the target platform's architecture and complex operation statements are refined to basic blocks, i.e., linear segments of instructions. In the ARMv4 instruction set, for instance, the control flow between basic blocks is implemented by conditional or unconditional branch operations like BGE (branch greater equal) or BLT (branch less than). Figure 4 shows our example as a control flow graph on the left and the set of basic blocks of instructions with their address space on the right. It is easy to see by the inspection of the control flow graph how comparison operators at C level correspond to branches in the ARMv4 opcode, i.e., that the [‘≤’➔‘>’] mutation operator directly corresponds to [BGE➔BLT] at the end of the two basic blocks.

Table 1. ARMv4 conditional instructions.

ARM Instruction	Condition (Symbol)	Condition (Binary Code)
BGE	≥	1010
BLT	<	1011
BGT	>	1100
BLE	≤	1101

Let us now consider the binary presentation of instructions, which is given in Table 1 for some ARMv4 instructions. We see that [BGE➔BLT] corresponds to [‘1010’➔’1011’], which refers to a bit flip of the forth bit in the binary code, for instance. Table 2 additionally gives an overview of ARMv4 mutation operators ordered by bit flips in their binary code representation. At this point we can now determine the probability of mutations at higher abstraction levels with respect to their relevance to bit flips. For our study that means, that, under the consideration of SEU tolerance, the first four mutations require special attention and measures already at higher abstraction levels where the others are less affected by SEUs. With that information we can thus consider the higher probability of this impact already from the very first beginning in the design process.

Table 2. ARMv4 mutation operators.

ARM Mutation Operator	Bit Flips
[BGE➔BLT]	1
[BLT➔BGE]	1
[BGT➔BLE]	1
[BLE➔BGT]	1
[BGE➔BGT]	2
[BGT➔BGE]	2
[BGE➔BLE]	2
[BLE➔BGE]	2
[BLT➔BLE]	2
[BLE➔BLT]	2
[BLT➔BGT]	3
[BGT➔BLT]	3

Therefore, we can already model and implement additional error correction measures especially dedicated to those four operators, which are then already covered by early testbenches. As the introduction of additional error correction measures always means higher costs in the software or hardware, this can avoid unnecessary costs for the introduction of error detections and corrections for events with very low probability. Additionally, this bottom up reflection can support the cross checking of the resistance of the design for coding errors vs. soft errors.

However, for an even more accurate analysis we also need information of the underlying hardware structures with concrete layout and process technology to extend our assumptions from single-word single-bit to single-word multi-bit upsets. Here, it is important to consider if register or memory cells have a linear or an interleaving layout for a higher multi-bit fault protection. In interleaving cell layouts, for instance, an event cannot have any impact on adjacent bits of a single word when it affects adjacent memory cells.

In the previous study we investigated just a simple example with focus on transient bit flips from SEUs. However, the basic principles are not limited to SEUs rather than apply to any bit flip scenario regardless they are due to soft or hard errors and regardless to the abstraction level.

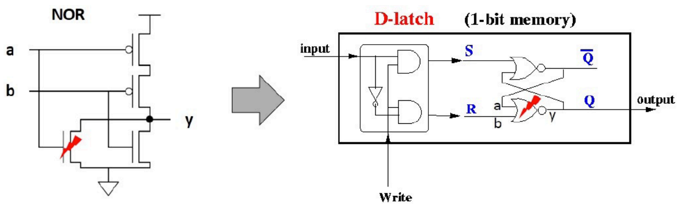


Figure 5. Transistor level bit flip.

Figure 5, for instance, gives the example of a permanent bit flip given by a stuck-at at gate and transistor level. Considering a simple 1-bit memory cell at transistor level, a stuck-at-short at a transistor of a NOR gate may stuck the gate output at ground, which corresponds to a stuck-at-0 at gate level, for instance. This gives an idea how to extend our bit flip considerations to hard errors and to gate and transistor level. Such extensions are also required when we either implement the specified function on an FPGA or by a digital comparator, respectively. Then we have correlations to the individual hardware components and layouts as the corresponding stuck-at or transient faults come with different probabilities compared to our previous study. Moreover, we can also implement that function by analog hardware with a comparison of voltage levels, for instance. However, that domain would require the application of inherently different means and measures for fault injection and testing considering inherently different analog artifact with different signal flips, which are not subject of this article.

IV. Summary and Conclusion

Efficient fault effect simulations are of utmost importance in the context of functional safety verifications enforced by international standards, where we studied the correlation of faults of different domains and at different abstraction levels. Our focus was on embedded software with a top down design flow: specification, modeling, source code implementation, and compilation to a binary code. We studied a simple example with fault injection by mutations for an operator and their correlations to SEU related bit flips in the opcode of a software instruction. Additional studies may cover bit flips in addresses or data sections of an instruction.

The main goal of this top down approach was to track mutations between different abstraction levels in order to derive the probability of the different mutations at binary and hardware level by a bottom up reflection. Such a combined top-down and bottom-up analysis is useful to identify individual locations, gaps, and additional necessary dedicated measures for error detection and correction, which propagate through all abstraction levels. As such, it indicates safety critical functions and operations, which require special attention from the very first beginning of the design process and which are candidates for reaching a higher safety level. It also identifies functions and operations with low fault injection probabilities, i.e., candidates where measures for error correction can be partially suppressed, which may save costs throughout the entire design process. Additionally, it comes with the advantage that testbenches at higher abstraction levels already cover mutations that do not only relate to coding bugs rather than additionally cover physical effects. This could be a major step forward as current risk analysis is most oftenly based on general uniformly distributed probabilities of bit flip patterns. However, we have demonstrated it just for a simple example and a very specific type of mutation. It certainly needs more studies and automation support to address more complex designs.

References

[1] International Electrotechnical Commission (IEC). IEC 60730: Automatic electrical controls - Part 1: General Requirements. International Standard, 2010.

[2] Radio Technical Commission for Aeronautics (RTCA). Design Assurance Guidance For Airborne Electronic Hardware. International Standard, 2000.

[3] International Standardization Organization (ISO). ISO 26262-5 Road Vehicles - Functional Safety - Part 5: Product development at the hardware level. International Standard, 2011.

[4] International Standardization Organization (ISO). EN ISO 13849-1 Safety of Machinery - Safety-Related Parts of Control Systems. International Standard, 2006.

[5] International Electrotechnical Commission (IEC). Iec 61025: Fault Tree Analysis (FTA). International Standard, 2006.

[6] International Electrotechnical Commission (IEC). IEC 60812: 2006. Analysis Techniques for System Reliability – Procedure for Failure Mode and Effects Analysis (FMEA). International Standard, 2006.

[7] European Aviation Safety Agency (EASA). EASA CM SWCEH 001 – Development Assurance of Electronic Hardware. EASA, August 2011.

[8] P.P. Shirvani, N.R. Saxena, E.J. McCluskey. Software-Implemented EDAC Protection against SEUs., IEEE Trans. Reliability, 49(3), IEEE Press, September 2000.

[9] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, M. Violante. Software-Implemented Hardware Fault Tolerance, Springer Verlag, Berlin 2006.

[10] R. A. DeMillo and A. J. Offutt. Constraint-Based Automatic Test Data Generation. IEEE Transactions on Software Engineering, 17(9), 1991.

[11] N.Jha and S. Gupta. Testing of Digital Systems. Cambridge University Press, Cambridge, UK, 2003.

[12] M. Hampton, S. Petithomme. Leveraging a Commercial Mutation Analysis Tool for Research. In Proc. of Testing Academic & Industrial Conference Practice and Research Techniques, Windsor, UK, 2007.

[13] M. Becker, D. Baldin, Ch. Kuznik, M.M. Joy, T. Xie, W. Mueller. Xemu: An Efficient Qemu Based Binary Mutation Testing Framework for Embedded Software. In Proc. of EMSOFT 2012, Tampere, FL, 2012.

Aging Aware Timing Analysis Incorporated Into a Commercial STA Tool

Shushanik Karapetyan and Ulf Schlichtmann
Institute for Electronic Design Automation,Technische Universität München, Munich, Germany
Email: Shushan.Karapetyan@tum.de and Ulf.Schlichtmann@tum.de

I. Introduction

Continuous scaling of transistor sizes to achieve low dynamic power, less area, and more speed has worsened the aging effects. Two dominant contributors affecting the transistor aging are Negative Bias Temperature Instability (NBTI) and Hot-Carrier-Injection (HCI) [1], [2]. Both of these mechanisms negatively impact the timing behavior of circuits. Traditionally, aging analysis has not been a part of the established circuit design flow and commercial tools do not yet support aging analysis on gate level, therefore aging analysis is not commonly available to industrial designers yet. Historically, to account for aging effects and variations, the common practice was to apply safety margins to the design. However, as the impact of aging effects increases, the necessity of their consideration in the design flow grows.

There is an active ongoing research in this area to explore and study the impact of aging mechanisms on circuit performance. Various aging models have been developed to predict device degradation. However, most of these models are bound to custom in-house scripts. This of course limits their usability. Moreover, some of the approaches require a modification of the original netlist [3].

We propose an automated methodology for integrating aging aware timing analysis into Synopsys PrimeTime to predict the combined impact of NBTI and HCI on a circuit performance. The approach utilizes the AgeGate aging aware logic gate model [4]. As a result it is now possible to utilize already available tool features for further design revision. For instance, information with regards to the critical path changes after aging aware timing analysis can be translated into a set of tighter timing constraints and applied to these paths. Logic optimization, in consideration of these new timing constraints will lead to an aging-robust design, thus increasing the lifetime of the circuit. In addition, the methodology does not require any modification of the design netlist. Results obtained from applying the method to various benchmark circuits are presented. These results demonstrate that aging is an important phenomenon that needs consideration in timing analysis and can be easily analyzed using commercial tools.

The fundamentals of the work have been peer reviewed and published at VLSI-DAT15 conference. The key differences of our approach compared to the state of the art are:

- For the calculation of aging induced performance degradation the proposed method does not require information about the system-level workload, but can work with reasonable assumptions about activity at the primary inputs if workload information is not available. Workload is defined by the portion of the lifetime a device is under a particular operating condition.
- It considers impact of both NBTI and HCI effects.
- The AgeGate model used in our work is based on traditional two-dimensional LUTs, therefore it can be used by an already existing analysis flow.
- No modification of the design netlist is required.
- The structure of the implemented algorithm is flexible, so in the future more aging effects can easily be integrated into it.

II. Proposed Methodology

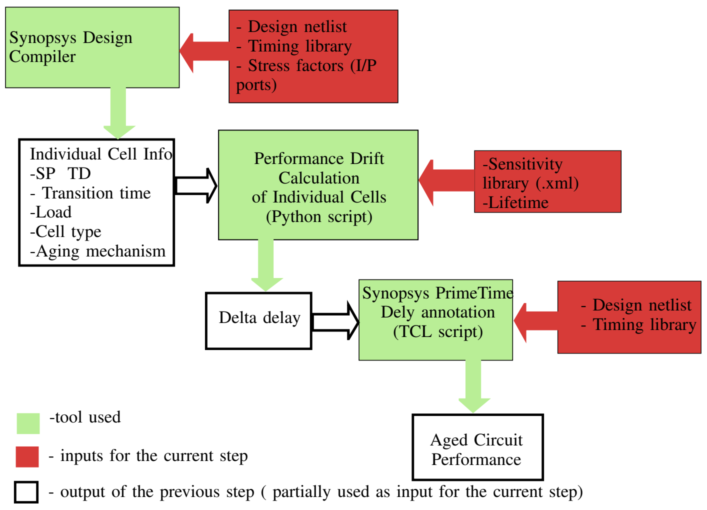
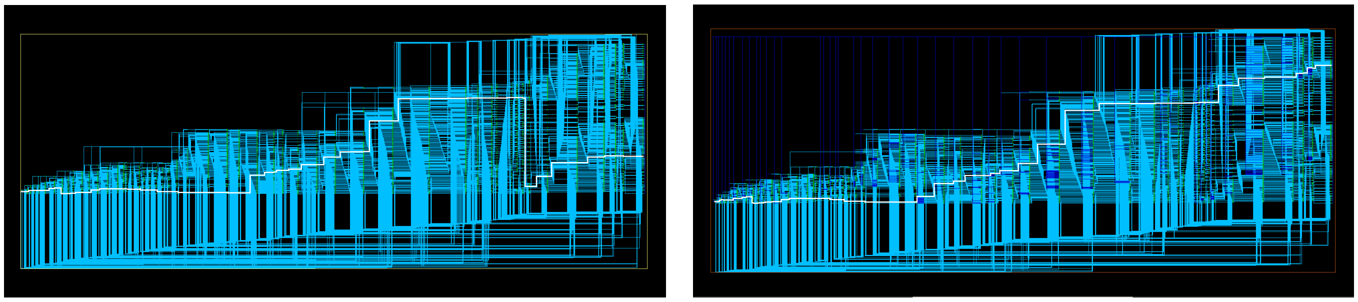

Synopsys Design Compiler
- Design netlist
- Timing library
- Stress factors (I/P ports)
Individual Cell Info
-SP TD
- Transition time
-Load
-Cell type
-Aging mechanism
Performance Drift Calculation of Individual Cells (Python script)
-Sensitivity library (.xml)
-Lifetime
Delta delay
Synopsys PrimeTime Dely annotation (TCL script)
- Design netlist
- Timing library
Aged Circuit Performance
-tool used
- inputs for the current step
- output of the previous step (partially used as input for the current step)

Fig. 1: Overview of the ASTA flow incorporated into Synopsys PrimeTime

The ASTA flow is similar to the traditional STA flow with the difference that the delay of the aged logic cell has to be considered instead of the fresh one. To obtain the aged gate delay, the AgeGate model is used. The model computes the degraded performance of the logic gate due to parameter drifts of the individual transistors caused by a specific aging mechanism. The transistor parameter drifts due to NBTI and HCI are calculated by technology specific degradation equations. The canonical gate model corresponds to a first-order Taylor approximation at the nominal gate performance q_{fresh}:

$$q_{aged} = q_{fresh} + \sum_{m \epsilon G} \sum_{p \epsilon P} x^{q}_{m,p} \Delta p_m \quad (1)$$

The overview of the ASTA flow integrated into Synopsys PrimeTime is presented in Fig.1. The amount of parameter drift is strongly dependent on the time a transistor is under a specific aging stress, therefore it is crucial to determine the fraction of


(a) Critical path of the fresh circuit

(b) Critical path of the aged circuit

Fig. 2: Critical path change of the c5315 benchmark circuit after aging

time that a transistor will be stressed under a specific aging effect during its lifetime. To do this, the first step in the flow is to obtain signal probabilities for all nets in a circuit. For this, first of all the workloads for all nets in the design have to be obtained. For this purpose, a system-level workload profile, if available, can be utilized. Alternatively, Signal Probability (SP) and Transition Density (TD) with reasonable assumptions can be applied to the primary inputs and be propagated throughout the circuit to its primary outputs. After signal probabilities have been obtained, the next step in the flow is to calculate the effective NBTI and HCI stress times for all transistors in the circuit. The stress time t_{stress} is determined by the following equation: The stress time (t_{stress}) is determined by the following equation:

$$t_{stress} = P_{stress} \cdot t_{lifetime} \tag{2}$$

where $t_{lifetime}$ is the defined lifetime of the circuit and P_{stress} is the probability that a transistor is stressed due to a specific aging effect. In order to accurately calculate the effective stress probability of a particular aging effect, information about the internal structure of a logic gate is also required, since it can happen that two transistors with equal signal probabilities have different stress probabilities because of their position in a logic cell. To calculate the P_{stress}, the individual cell information has to be extracted from PrimeTime. This information includes: cell type, signal probabilities, transition time, and output load. Now, that the P_{stress} is available, the next step is to calculate the parameter drift caused by a specific aging effect. The calculation is done by means of technology specific degradation equations for NBTI and HCI. The use profile information (supply voltage, temperature, and lifetime) along with the output of the previous step, serves as an input to the degradation equations. Once the parameter drifts are computed, the delay degradation is obtained from the AgeGate model by combining the information about the parameter drift with the pre-characterized sensitivity coefficients. Finally, in the last step the portion of aging induced delay degradation obtained in the previous step is added to the individual gate delays of the design. The delay annotation is done by individually applying timing derates to the logic cells in the design. Finally, these annotated gate delays enable PrimeTime to perform ASTA considering one or more aging mechanisms.

Design	Fresh delay [ns]	NBTI [%]	HCI [%]	BOTH [%]	Run time [s]
c17	0.1312	3.0	2.58	5.58	2
c880	1.1212	6.12	2.86	8.92	6
c1355	1.385	7.33	2.77	10.10	7
c1908	1.996	5.75	2.88	8.64	9
c2670	2.164	8.80	2.73	11.46	14
c3540	2.66	9.27	2.70	11.90	18
c5315	2.266	8.05	2.91	10.96	29
c6288	5.65	4.88	2.76	7.64	31
c7552	1.736	7.30	2.88	10.18	35

TABLE I: Critical path delay degradation of ISCAS'85 circuits considering NBTI, HCI or BOTH effects together

III. Results

The results presented here are obtained by applying the proposed methodology to various benchmark circuits. The results of the analyses show that NBTI is the dominant aging effect. However, the impact of HCI on circuit degradation must not be neglected. Based on the analysis results, the delay degradation of a circuit can reach up to 11.90%. Because different paths in a design have different workload and signal probability, they can age at different rate. This may lead to a critical path change of a design. Such a case is presented in Fig. 2. It shows the critical paths of the c5315 benchmark circuit before (2(a)) and after (2(b)) aging. Because of the integration of aging analysis into a commercial framework, the corresponding paths are readily visualized to the designer in his regularly used tools.

References

[1] M. A. Alam and S. Mahapatra, "A comprehensive model of pmos nbti degradation," in *Microelectronics Reliability*, vol. 45, no. 1, Jan 2005, pp. 71–81.

[2] P. Fang, J. Tao, J. Chen, and C. Hu, "Design in hot-carrier reliability for high performance logic applications," in *CICC*, May 1998, pp. 525–531.

[3] F. Firouzi, S. Kiamehr, M. Tahoori, and S. Nassif, "Incorporating the impacts of workload-dependent runtime variations into timing analysis," in *DATE*, March 2013, pp. 1022–1025.

[4] D. Lorenz, M. Barke, and U. Schlichtmann, "Aging analysis at gate and macro cell level," in *ICCAD*, Nov 2010, pp. 77–84.

An FPGA-based Testing Platform for the Validation of Automotive Powertrain ECU

L. Sterpone, D. Sabena, L. Venditti
Dipartimento di Automatica e Informatica
Politecnico di Torino
Torino, Italy

Abstract*—Over the past decade, the complexity of electronic devices in the automotive systems is increased significantly. The today high level vehicles include more than 70 Electronic Control Units (ECUs) aimed at manage the powertrain of the vehicle, and improve passengers comfort and safety. ECU microcontrollers aimed at the control of the fuel injection system have a key role. In this paper we present a new FPGA-based platform able to supervise and validate Commercial-Off-The-Shelf timer modules used in today state-of-the-art software applications for automotive fuel injection system with an accuracy improvement of more than 20% with respect to traditional approach. The proposed approach allows an effective and accurate validation of timing signals and it has two main advantages: can be customized with the exact timing module configurations to meet the exigency of new tests and allows effective modularity design test. As case study two industrial Time Modules manufactured by Freescale and Bosch have been used. The experimental analysis demonstrate the capability of the proposed approach providing a timing and angular precision of 10 ns and 10^{-5} degrees respectively.

Keywords—Automotive; FPGA; ECU validation; eTPU; GTM.

I. Introduction

The today automotive development processes are characterized by an increasing complexity in mechanic and electronic. However, electronic devices have been the mayor innovation driver for the automotive systems in the last decade [1][2]. In this context, the requirements in terms of comfort and safety lead to an increasing number of on-vehicle embedded systems, with more and more software-dependent solutions using several distributed Electronic Control Units (ECUs).

Sophisticate engine control algorithms require performance enhancement of microprocessors to satisfy real-time constraints [3]. Moreover, the code generation, the verification, and the validation of the code itself, become key part in the automotive domain: the software component development processes have to be as efficient and effective as possible. Moreover, without a reliable validation procedure, the automotive embedded software can lead to a lot of errors and bugs, and decrease the quality and reliability of application software components.

Electronic devices managing the fuel injection in today engines have a key role, in order to guarantee efficient and powerful vehicles [4]. The recent research works in the area are aimed at reducing the fuel consumption, while maximizing the power conversion and reducing air pollution emissions [5][6]. As reported in [7], a major challenge being faced in diesel technology is meeting current and future emission requirements without compromising fuel economy. Clearly, these goals could be reached through improvements in the engine electronic management. In this context, efficient fuel injection control is required [8].

Given this behavior, microcontrollers devoted to the engine management contain specific timer modules aimed at generating, among the others, the signals used by the mechanical parts controlling the fuel injection in the cylinders[9]. The scope of these timers is to provide the real-time generation of the signals, ensuring an efficient engine behavior. In order to achieve the correct level of synchronization between the engine position and the generated fuel injection signals, the automotive timer modules typically receive a set of reference signals from the engine; the most important are the *crankshaft* and the *camshaft* [10]. These signals are used to detect the current engine position, i.e., the angular position of the cylinders within the engine [11]. A precise detection of these information items represents a key point for all the electronic engine management [12].

The correct programming of the timer modules is a key aspect in the automotive domain, due to the complexity of its programming code, and of the applications they have to manage. Consequently, efficient and precise validation methods and platforms are required. The current main methods to validate automotive engine applications are based on models [8][13][14], or on ad-hoc special purpose test equipment available on the market [15]. As timer module become more advanced, it raises the cost associated with validating these new modules, since extremely complex and expensive equipment must be adopted and traditional equipment are no longer able to keep up with constantly changing requirements of these systems.

In this paper, we present a new FPGA-based validation platform aimed at the validation of the applications running in the real-time timer modules used in the today vehicles. The purpose of this platform is to provide the developers of automotive applications with a flexible and efficient architecture able to effectively validate the code running in the most popular timer modules. More in particular, the proposed platform has the capability of generating the engine reference signals (i.e., the crankshaft and camshaft reference signals) that are typically used by the automotive microcontrollers to generate the fuel injection signals, and acquiring the signals generated by the timer module under test, verifying the synchronization between these signals and the provided engine reference signals. The proposed platform is useful to validate the functioning of several timer modules running in different engine configurations (e.g., with different profile of the crankshaft and camshaft signals). The platform can be customized with the exact instrument modules to meet the exigency of new test, it has flexible functionalities for diverse test bench purposes. Finally, it allows effective testing of modular designs and if compared with traditional approaches used to test state-of-the-art timer modules, it has an accuracy improvement of more than 20% [15] providing a timing and angular precision of 10 ns and 10^{-5} degrees respectively.

As case study, we use two important timer modules used today in the automotive domain: the *Enhanced Time Processor Unit (eTPU)* developed by Freescale [16][17], and the *Generic Timer Module (GTM)* developed by Bosch [18]. We choose these two modules since they are used, among the others, for the generation of the fuel injection signals in several engines; moreover, eTPU and GTM represent a good set of benchmarks, since the way in which they manage the automotive applications are different: in fact, in the eTPU several software routines share the same processing unit, while in the GTM several tasks can be directly managed by hardware parallel processing units.

The acquired results demonstrate the validity of the proposed approach, since using the proposed platform, it is possible to verify the synchronization between the inputs and the generated fuel injection signals with a very high degree of precision. Moreover, the proposed platform is able to verify the signals synchronization both in static engine conditions (i.e., constant engine speed), and in dynamic conditions (i.e., variable engine speed).

The rest of the paper is organized as follow: Section II overviews some previous work in the area of the automotive electronic control development and validation; Section III describes the main task performed by the today automotive timing modules embedded in the recent microcontrollers; Section IV details the proposed validation platform, while in Sections V the experimental results are shown. Finally, Section VI draws some conclusions and outlines the future works.

II. RELATED WORKS

With the development of electronic technology and the application of control theory in the automotive control [19], many research works have been developed with the purpose of improve the control of the fuel injection. The motivation of these research works is that, nowadays, the fuel injection system is the most important part of diesel engines, and its working state directly influences the performance, the consumption and the air pollution, as documented in [4][8][12].

In [4] the authors present a new fuel injection intelligent control system, designed to improve the testing accuracy. The proposed system can automatically test the state of the injection pump, and it obtains the all parameters of the fuel injection system without human intervention by the use of PC and AT89C52 single chip microcomputer. Such system is designed and realized on the SYT240 fuel injection system test platform, which can automatically fetch and display the main parameters. Although the approach presented seems to be promising, it is strongly based on the usage of a dedicated test platform.

In [5] the authors face the problem of improving the accuracy of the engine control electronic, and they affirm that one potential way to do this is by use real-time in-cylinder pressure measurements. Consequently, the authors propose an approach that derives the pressure information from the measurement of the ordinary spark plug discharge current. The motivation of this work is that, by monitoring the pressure of each cylinder, the electronic engine control can be optimized in terms of fast response and accuracy, thus enabling online diagnosis and overall efficiency improvement.

Another research work addressing the usage of cylinder pressure-based combustion control is presented in [7], where the authors explain that in case of multiple fuel injections, the timing and the width of the fuel injection pulses need to be optimized. More in particular, this paper presents several methods in which the cylinder pressure signal is used for multiple-pulse fuel injection management for a diesel engine capable of running in low-temperature combustion modes.

In[6] the authors explain that it is important to avoid discrepancies between the fuel amounts injected into the individual cylinders, in order to avoid excessive torsional vibrations of the crankshaft. Consequently, the authors present a general adaptive cylinder balancing method for internal combustion engines; the proposed algorithm uses online engine speed measurements. The motivation of this work is that due to varying dynamics and ageing of components of the fuel-injection systems, there may be a significant dispersion of the injected fuel amounts.

In order to implement all the fuel injection optimization methods proposed above, in the real engine behavior, the engine angular position has to be identified as precisely as possible. In [16] the authors present an example of engine position identification by using the eTPU module embedded in the MPC5554 microcontroller.

Another work addressing the problem of a precise angular engine position detection is reported in [12]; more in particular, the authors explain that, due to mounting and packaging tolerances, the magnetic field at the sensors position varies, resulting in angular measurement. Mounting and packaging tolerances cannot be avoided; consequently, the authors propose a compensation method based on a new filter structure.

Summarizing, several research works have been developed in the last decade in order to optimize the fuel injection systems of the today vehicles. In order to achieve this goal, the usage of specific timer modules, i.e., the eTPU and the GTM, is today required; the main tasks typically managed by these modules are acquire in a very precise way the engine angular position, and generate, among the others, the signals aimed to control the cylinders fuel injection. To do this, these modules have to be configured, using their specific programming code; this task is a difficult and an important part of a development of the fuel injection control systems, since a small error can cause relevant problems in the engine behavior, thus compromising the efficiency of the entire system.

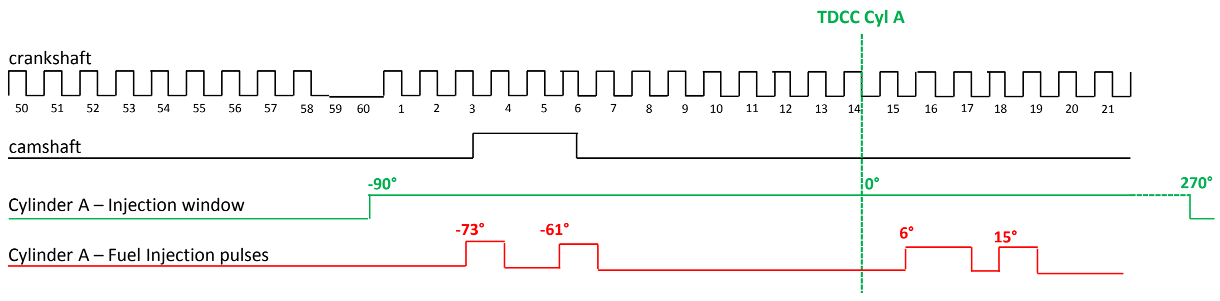


Fig. 1. Example of the main signals received and managed by the automotive timer modules, in order to efficiently supervise the engine behavior.

In this paper we propose an FPGA-based validation platform able to emulate the engine behavior (i.e., generate the crankshaft and the camshaft signals), and to acquire the fuel injection signals generated by the timer module under test; using this platform, it is possible to validate the timer module in a very precise way since the precision of the data acquisition is about 10^{-5} engine angular degrees, which is an improvement of more than 25% with respect to traditional methods [14][15]. The effective synchronization validation between the input and the output signals is performed by the developed software framework which compares the data achieved during the experimental analysis with the expected ideal values. To the best of our knowledge, this is the first work addressing the generation of a flexible and low-cost validation platform for the today automotive microcontrollers. The main contribution of this work is to provide at the developers of the automotive applications a precise and flexible validation platform, useful to check the correctness of the developed software routines, thus ensuring an efficient system development. Moreover, using this platform it is possible to check the functioning of the real microcontroller, avoiding the unexpected misbehaviors due to the model-based validation of the developed applications.

III. Timer Modules in Automotive Applications

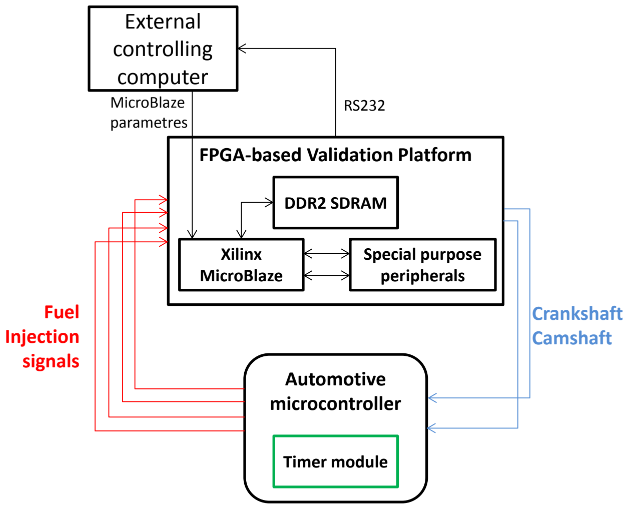
The today automotive microcontrollers contain specific timer modules to manage the engine signals. In Fig.1, the most important signals related to the cylinders fuel injection are shown. All the main tasks performed by the automotive microcontrollers are based on a precise detection of the engine angular position (i.e., the precise position of the cylinders with respect to the crankshaft). This is done using the two reference signals coming from the engine, i.e., the crankshaft and the camshaft. The crankshaft, typically, is a square wave signal, where each falling edge transition represents a partial rotation of the crankshaft. For example, if the crankshaft phonic wheel [20] is composed of 60 teeth, each falling edge transition of the crankshaft signal indicates a rotation of 6°. Moreover, in a determinate position of the crankshaft signal, a *gap* (i.e., a missing tooth) is present: this gap is used as reference point to understand the correct engine angular position [25]. On the other side, the camshaft is a signal composed of few pulses synchronized with the crankshaft. Since the engines addressed in the context of this paper are 4-stroke engines, the complete 4-stroke sequence (i.e., intake, compression, power, and exhaust) takes two full rotations of the crankshaft. By only looking at the crankshaft signal, there is no way to understand if the crank is on its intake-compression rotation or on its power-exhaust rotation. To get this information, the camshaft signal is required; moreover, due to the 4-stroke configuration, the camshaft rotates at half the crankshaft speed (a rotation of 360° of the camshaft implies a rotation of 720° of the crankshaft); consequently, a signal generated once per rotation of the camshaft is sufficient to supply the required information. According to the features of these signals, the *Top Dead Cylinder Center (TDCC)* for each considered cylinder is identified [21]. The fuel injection pulses are electronic pulses that act on the fuel injector of each cylinder. These pulses have to be generated in a very precise angular position, where the reference point is the TDCC. The range in which these pulses can be generated is called *Injection Window (IW)*; typically, the width of the IW is 360°. In the context of this paper, we consider a maximum number of injection pulses equal to 16. The angular position of the beginning of an injection pulse is called *Start Of Injection (SOI)*, or *Start Angle*; moreover, the injection pulses can be programmed using a temporal displacement between them (*Dwell*). Finally, in the typical automotive applications, timer modules generate, also, a sequence of high frequency *pulse width modulation (PWM)* pulses, in order to trigger other engine sensors.

The generation of the fuel injection pulses is not the main goal of this paper and, consequently, we don't explain other details about this topic. The method proposed in the next section is focused on the verification of the precision and of the synchronization of the generation of the fuel injection pulses.

IV. The Proposed Validation Platform

The main purposes of the proposed platform are generate the engine reference signals, and acquire the fuel injection pulses, generated by the microcontroller under test, correlated with the provided engine reference signals itself. In Fig. 2 the overview of the proposed validation behavior is shown. It is composed of the FPGA-based validation platform and of the external controlling computer.

The validation platform is composed of the 32-bit Xilinx MicroBlaze processor core [22] and of a set of special purpose DSP peripherals designed to support the validation of the signals generated by the timer module under test. Both the MicroBlaze processor and the DSPs are implemented in a FPGA device. The communication between the processor and the DSP are managed using a Processor Local Bus (PLB) interface. An external controlling computer is directly connected to the processor, in order to provide the input parameters required to the generation of the engine signals. Moreover, a RS232 interface is used to save the acquired data into a database contained in the external computer itself.



External controlling computer
MicroBlaze parametres
RS232
FPGA-based Validation Platform
DDR2 SDRAM
Xilinx MicroBlaze
Special purpose peripherals
Fuel Injection signals
Crankshaft Camshaft
Automotive microcontroller
Timer module

Fig. 2. The overview of the proposed validation platform

Two are the most important DSP peripherals developed in this context: the crank/cam generator which consists on the module used to generate the crankshaft and the camshaft signals according to the user parameters, and the measurement module which is sampling and storing the signals provided by the timing module under test.

A. *The Crankshaft and Camshaft DSP peripheral*

In Fig. 3 the composition of the crankshaft and camshaft DSP peripheral is shown. It is composed of three main sub-modules: *clock_div*, *selector_crank*, and *selector_cam*. The goal of this peripheral is to generate the crankshaft and camshaft signals in two different ways, according to the user requirements: in the former, the signals have to emulate the signals of an engine running with constant *rounds per minutes (rpm)*, while in the second case these signals have to emulate the dynamic behavior of an engine, i.e., non-constant rpm.

The clock_div sub-module receives in input from the MicroBlaze processor (through the use of *slave registers*) the 32-bit values *Delta_period* and *Num_cycles*; the former is a timeout value (expressed in number of clock cycles) in which the period of the crankshaft teeth have to remain constant. When this time is elapsed, the *request signal* is raised in order to communicate at the MicroBlaze processor (through the use of an interrupt) that the new speed parameters can be sent; this mechanism allows to generate dynamic or static crankshaft and camshaft signals. The second value received by the peripheral (i.e., Num_cycles) represents, instead, the actual speed of the engine: in particular, it is the duration, expressed in number of clock cycles, of half period of the crankshaft signal. The internal circuitry (i.e., the selector crank sub-module) causes a toggle of the output crankshaft signal every Num_cycles clock cycles. A similar approach is used to generate the output camshaft signal: the signal *Cam Enable 2*, generated by the selector crank sub-module, reports at the selector cam sub-module when toggle the output camshaft signal. This has been done to ensure a very precise synchronization between the crankshaft and the camshaft signals generated by this peripheral.

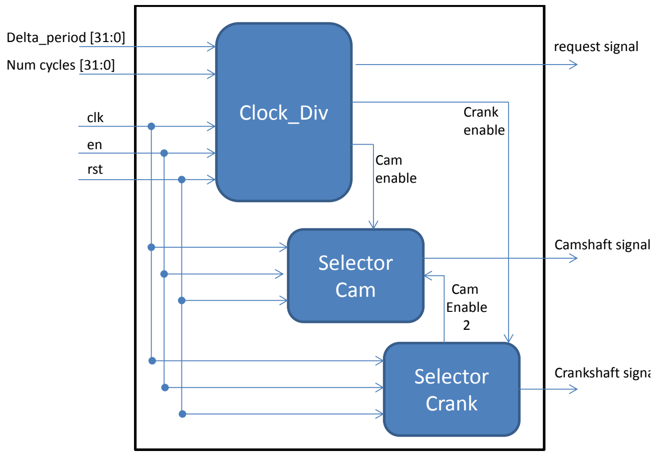


Fig. 3 The crankshaft and camshaft generator peripheral.

B. *The measure DSP peripheral*

The measure peripheral receives in input the injection pulses and the PWM pulses signals, both generated by the timer module under test; using as absolute reference the crankshaft and the camshaft signals (generated by the specific peripheral explained in the previous section) it performs the required measurements of the input signals. This peripheral is able to measure the signals of one cylinder at a time.

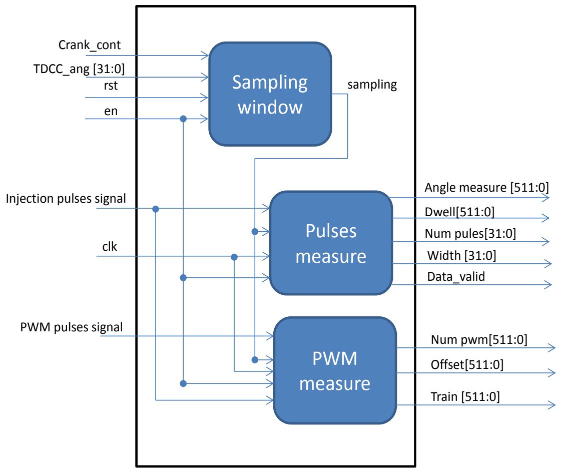


Fig. 4 The measure peripheral.

In Fig. 4, the main sub-modules composing the measure DSP peripheral are shown. The *sampling window* sub-module triggers the other two sub-modules: it receives in input (*TDC_ang* signal) from the MicroBlaze processor (through the use of a slave register) the number of the crankshaft tooth falling edge corresponding to the angular value of the TDCC referred to the cylinder to be monitored. It also receives in input the crankshaft signal generated by the other peripheral (*crank_cont* signal). By counting the falling edges of the crankshaft signal, the sampling window sub-module is able to produce in output the *sampling* signal, that remains active (i.e., logic value equal to 1) for 360° according to the programmed cylinder. This signal is connected to the *pulse measure* and to the PWM *measure* sub-modules, in which it is used as "clear" signal for the internal counters.

The pulses measure sub-module counts the number of incoming pulses (*injection pulses signal*) and exploiting a couple of latches, it stores in a set of internal signals the time stamps at which the rising and falling edges occur. To do this, an internal counter is used, where the frequency of the counter itself is the same of the frequency of the clk signal; moreover, this counter is reset at the beginning of each injection window. As soon as the sampling signal became 0, all the stored results are transferred to the respective 512-bit outputs. The length of the output signals is due to the fact that inside a programming window, there should be at most 16 injection pulses; consequently, we have to store 16 parameters, each one represented with 32 bit. The output data of this sub-module are: (1) *angle measure*, which contains the measured start angle time stamps of the injection pulses; (2) *dwell*, which contain the measured time between the current pulse and the previous one; (3) *num pulses*, which indicates the total number of detected pulses inside the injection pulses; (4) *width*, which contains the measured time duration of each injection pulse; and (5) *data valid*, that signals when all the measured values have been copied in the output signals; this value is used to signal at the MicroBlaze processor (through an interrupt) that the current data could be transferred in the SDRAM.

A similar approach has been used for the PWM measure module, in which the features of the PWM signal generated by the timer module under test are measured. This module is activated once the injection pulses signal is detected. The output data of this module are: (1) *offset*, which contains the measured offset from the fuel injection pulses falling edge to the first PWM rising edge; (2) *train*, which contains the measured PWM train duration (in terms of number of clock cycles), from the first rising edge to the last falling edge; and (3) *num_pwm*, which contains the number of counted PWM pulses.

C. *The MicroBlaze processor tasks*

Considering the special purpose peripherals described in the previous sections, the MicroBlaze processor has three main tasks. The first is to provide the crankshaft and camshaft generator peripheral with the data about the features of the camshaft and crankshaft signals that have to be generated, i.e., the crankshaft period and the dynamic of the crankshaft signal itself. The second performed task is, whenever the data valid signal is received from the measure peripheral, transfer the acquired data (contained in the output signals of the measure peripheral) into the SDRAM; this allows to acquire the same data (about the fuel injection and the PWM signals) in different time instants, in order to characterize in a very precise way the measured signals produced by the microcontroller under test. Finally, when the required number of measurements have been acquired, the MicroBlaze processor takes care of sending (through a RS232 interface) the data contained in the SDRAM at the external controlling computer.

D. *The output data format*

All the measurements acquired by the peripherals described in the above sections, are based on temporal intervals and are contained in a file that we call *data_raw.txt*.

Since the main purpose of the proposed validation platform is to verify the synchronization of the fuel injection pulses with respect to the engine reference signals, we need to translate the acquired data about the pulses start angle from the FPGA time domain (i.e., the number of clock cycles measured by the peripheral) to the engine angle domain. To do this, the following formula has been used:

$$StartAngle[\deg] = \frac{(6 * StartAngle[ClockCycleNum] * RPM)}{ClockCyclePeriod[s]}$$

where the degrees of each crank falling edge transition are 6, and RPM are the current rounds-per-minute of the engine. Clearly, this formula can be used in the case of constant engine rpm; in case of dynamic engine behavior, the formula has to take in consideration the different RPM values during the measurements interval.

V. EXPERIMENTAL RESULTS

In Fig. 5 the experimental flow is shown. A Matlab parser translates the data acquired by the proposed validation platform. Then, these data are compared with the file containing the expected values (called *ideal_values.txt*); this file is generated by an ideal data generator written in C language. As a result, several report files are obtained; these files contain the details about the measures, including the average error, the maximum error and the standard deviation of each feature of each injection pulse generated by the timer module under test. As case study, we used two timer modules:

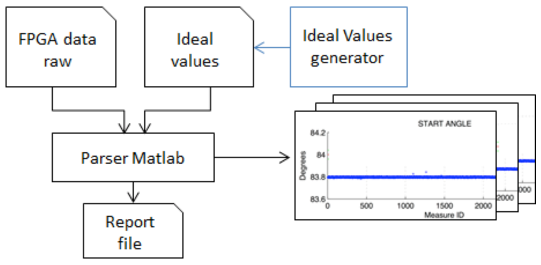


Fig. 5 The experimental flow.

the eTPU and the GTM; in the following section we explain the main features of these modules and the obtained measurements results.

A. *Enhanced Time Processor Unit (eTPU)*

The Enhanced Time Processor Unit (eTPU) [16][17] by Freescale, is an effective timing co-processor available in the automotive domain; it is used to efficiently managed I/O processing in advanced microcontroller units. From a high level point of view, the eTPU has the characteristics of both a peripheral and a processor, tightly integrated between each other [23]; essentially, it is an independent microcontroller designed for timing control, I/O handling, serial communications, and engine control applications [17]. More in particular, the eTPU is mainly used to decode the engine angular position, and, consequently, to control actuators such as the fuel injectors and the spark plugs, thanks to the high flexibility of the dedicated programmable hardware.

In the context of this paper, we used the eTPU module embedded in the microcontroller SPC5644AM; moreover, we used the automotive functions set available in [24].

B. Generic Timer Module (GTM)

The Generic Timer Module (GTM) [18] is a recent hardware module provided by Bosch. It is composed of many sub-modules with different functionalities. These sub-modules can be interconnected together in a configurable manner in order to obtain a flexible timer module for different application domains. The scalability and configurability is reached by means of the architectural structure of the module itself: a set of dedicated sub-modules is placed around a central routing unit, which is able to interconnect the sub-modules according to the programmed configuration specified in the running software [18]. The GTM is designed to run with a minimal CPU interaction and to unload the CPU itself from handling frequent interrupts service requests.

In the context of this paper, we used the GTM module embedded in the microcontroller SPC574K72; moreover, we directly implemented a set of automotive functions useful to generate the fuel injection pulses using only the GTM module.

C. The used Xilinx FPGA board

In order to implement the proposed validation platform, the Xilinx Virtex-5 XC5VLX50T FPGA has been used. This FPGA is embedded in a Digilent Genesys board. The working frequency of the MicroBlaze processor implemented in the FPGA itself in 100MHz; also the clock frequency provided at the special purpose peripherals is 100MHz. This allows us to obtain time measurements with a precision of 10ns, and angular measurements with a precision of 10^{-5} degrees.

D. Main obtained results

The main purpose of this section is to give the reader an idea of the effective features of the proposed validation platform, highlighting its capability of making measures with extreme precision.

In Fig. 6 two graphs are shown: the first (Fig. 6.a) reports the measurements of the width of an injection pulse, while the second (Fig. 6.b) shows the measurements of the PWM offset; in this case the module under test is the GTM. In Fig. 7 a graph reporting the measurements of the start angle of an injection pulse generated by the eTPU module is shown. By looking at this graph, it is possible to understand if the injection pulse is generated in the correct position, i.e., in a determinate angle position, according to the crankshaft and the camshaft signals. In this case, the precision of the measurements is 10^{-5} degrees.

As it is possible to notice by the graphs reported in this section, using the proposed validation platform it is possible to understand if the injection pulses are correctly generated. This allows to understand if the software applications running in the timer module are correct (ensuring a real-time behavior) or contain software bugs. Using this platform, thus, the developers of automotive applications can verify if the applications that they are writing efficiently manage the fuel injectors.

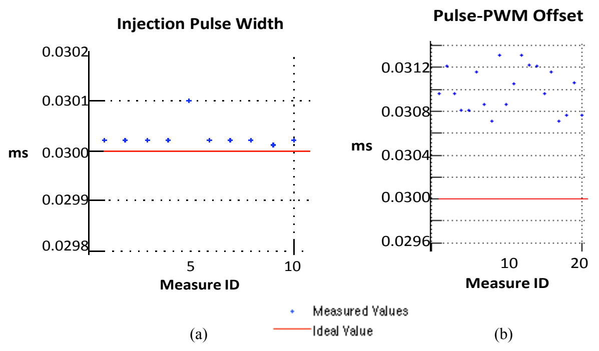


Fig. 6 Measures of injection pulse width (a), and of the offset between the injection pulse and the pwm signal (b); both the signals are generated by the GTM.

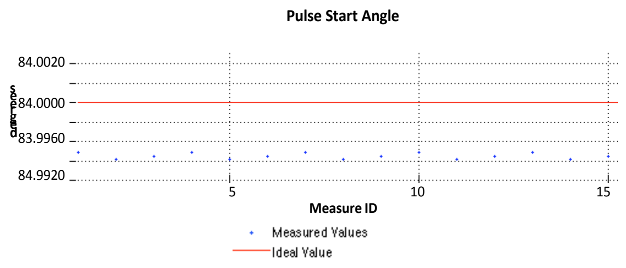


Fig. 7 Measures of the StartAngle of an injection pulse generated by the eTPU module.

VI. Conclusions and Future Works

In this paper we present a new platform for the validation of timer modules used in automotive applications. The high flexibility, combined with the capability of extreme precise measurements, make the platform very suitable to be used by the developers of automotive applications during the software development. As case study, we used two important timer modules employed in the today vehicles.

As future work, we plan to extend the proposed FPGA-platform in order to inject faults in the input signals provided at the timer module under test, checking the correctness and the synchronization of the generated output signals.

References

[1] E. Armengaud, A. Steininger, M. Horauer, “Towards a Systematic Test for Embedded Automotive Communication Systems,” IEEE Transactions on Industrial Informatics, vol. 4, n.3, 2008.

[2] M. Steger, C. Tischer, B. Boss, A. Muller, O. Pertler, W. Stolz, S. Feber, “Introducing PLA at Bosch Gasoline Systems: Experiences and Practices,” Springer Software Procuct Lines, Lecture Notes in Computer Science, vol. 3154, pp. 34 – 50, 2004.

[3] Y. Kanehagi, D. Umeda, A. Hayashi, K. Kimura, H. Kasahara, “Parallelization of automotive engine control software on embedded multi-core processor using OSCAR compiler,” IEEE Cool Chips XVI, p. 1 – 3, 2013.

[4] F. Juan, M. Xian-Min, “Research on fuel injection intelligent control system,” IEEE Conference on Industrial Electronics and Applications (ICEA), pp. 2782 – 2785, May 2009.

[5] A.D. Grasso, S. Pennisi, M. Paparo, D. Patti, “Estimation of in-cylinder pressure using spark plug discharge current measurements,” European Conference on Circuit Theory and Design (ECCTD), pp. 1 – 4, 2013.

[6] F. Ostman, H.T. Toivonen, “Adaptive Cylinder Balancing of Internal Combustion Engines,” IEEE Transacrions on Control System Technology, vol. 19, n. 4, pp. 782 – 791, 2011.

[7] I. Haskara, W. Yue-Yun, “Cylinder Pressure-Based Combustion Controls for Advanced Diesel Combustion With Multiple-Pulse Fuel Injection,” IEEE Transactions on Control Systems Technology, vol. 21, n. 6, pp. 2143 – 2155, 2013.

[8] Q. Lui, H. Chen, Y. Hu, P. Sun, J. Li, “Modeling and Control of the Fuel Injection System for Rail Pressure Regulation in GDI Engine,” IEEE/ASME Transactions on Mechatronics, vol. 19, n.5, pp. 1501 – 1513, 2014.

[9] J. Larimore, E. Hellstrom, S. Jade, J. Li, “Controlling Combustion Phasing Variability with Fuel Injection Timing In a Multicylinder HCCI Engine,” American Control Conference (ACC), pp. 4435 – 4440, 2013.

[10] T. A. Johansen, O. Egeland, E. A. Johannessen, R. Kvamsdal, “Free-piston diesel engine timing and control - toward electronic cam- and crankshaft,” IEEE Transactions on Control System Technology, vol. 10, n. 2, pp. 177 – 190, 2002.

[11] S. Hainz, E. Ofner, D. Hammerschmidt, T. Werth, “Position Detection in Automotive Application by Adaptive Inter Symbol Interference Removal,” IEEE 5th conference on Sensors, pp. 1103 – 1106, 2006.

[12] S. Hainz, D. Hammerschmidt, “Compensation of Angular Errors Using Decision Feedback Equalizer Approach,” IEEE Sensor Journal, vol. 8, n. 9, pp. 1548 – 1556, 2008.

[13] F. Li, T. Shen, X. Jiao, "Model-based design approach for gasoline engine control Part I: Modeling and validation," 32nd Chinese Control Conference (CCC), pp. 7774 - 7779, 2013.

[14] E. Alabastri, L. Magni, S. Ozioso, R. Scattolini, C. Siviero, and A. Zambelli, “Modeling, analysis and simulation of a gasoline direct injection system,” in *Proc. 1st IFAC Symp. Adv. Automot. Contr.*, 2004, pp. 273–278, 2004.

[15] National Instrument Data-Sheet, “Counter/Timer Overivew”, pp. 386 – 393, 2013.

[16] W. Dafang, L. Shiqiang, J. Yi, Z. Guifan, “Decoding the Engine Crank Signal Referring to AUTOSAR,” Intelligent Computation Technology and Automation (ICICTA), 2011 International Conference on, pp. 616 – 618, March 2011.

[17] Enhanced Time processor Unit (eTPU). Available [Online]: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=eTPU

[18] Generic Timer Module (GTM) Product Information. Available [Online]: http://www.bosch-semiconductors.de/media/en/pdf_1/ipmodules_1/timer/bosch_product_info_gtm_ip_v1_1.pdf.

[19] A. Ohata and K. R. Butts, “Improving model-based design for automotive control systems development,” in *Proc. 17th World Congr.*, ,pp. 1062–1065, 2008.

[20] X. Ying, Y. Qiangqiaang, L. Fuyuan, “Research of Crankshaft Grinding Wheel Dresser Based FANUC NC System,” 3rd International Symposium on Information Processing (ISIP), pp. 189 – 192, 2010.

[21] T. Yamanaka, M. Esaki, M. Kinoshita, “Measurement of TDC in Engine by Microwave Technique,” IEEE Transaction on Microwave Theory and Techniques, vol. 33, n. 12, pp. 1489 – 1494, 1985.

[22] Xilinx MicroBlaze details. Available [Online]: http://www.xilinx.com/tools/microblaze.htm

[23] C. Rodrigues, “A case study for Formal Verification of a timing co-processor,” 10th Latin American Test Workshop (LATW), pp. 1-6, 2009.

[24] Freescale automotive funcions set. Available [online]: http://www.freescale.com/webapp/etpu/

[25] Freescale Application Note. Available [Online]: http://cache.freescale.com/files/32bit/doc/app_note/AN3769.pdf

Incremental System Design with Cross-Layer Dependency Analysis

Mischa Moestl and Rolf Ernst
Institut für Datentechnik und Kommunikationsnetze (IDA)
Technische Universität Braunschweig
Braunschweig, Germany
{ moestl | ernst }@ida.ing.tu-bs.de

I. Introduction & Motivation

Safety-critical systems usually require certification or adherence to specific safety standards imposed by the application domain. Common standards such as the general IEC 61508 [1] which has derivatives in different domains, e.g. ISO 26262 [2] in the automotive domain, demand for applications with different assurance levels to be integrated into the same system, either that all applications be certified to the highest applicable level or that *sufficient independence* between the elements of a system can be established. In general this means that interference between two functions must not affect any safety related property of the entire system.

In order to argue *sufficient independence*, the standards suggest two generic methods to investigate a system. First of all, there is the top-down approach of Fault Tree Analysis (FTA) that is standardized in [3] that tries to investigate possible root causes for specified system failures. The second method, referred to in e.g. [1], is the bottom-up method of Failure Modes and Effects Analysis (FMEA) [4]. However, FMEAs and FTAs require the full knowledge about the chain of effects in a system, i.e. it is not only necessary to know that something happens; it is inevitable to know how it is achieved. For example, not only the fact that a message is exchanged between two components A and B is relevant, it is necessary to exactly know how and by which means this is achieved. The reason why this is required, and why FMEA is a powerful method to assess designs for safety, is that all possible influences, i.e. failure modes, which can have an influence on the nominal chain of effects, are studied extensively. Yet, this requires that the full chain of events is known a priori in order to conduct an FMEA that thoroughly investigates whether a design is safe for the function under consideration. This is a clear limitation of the applicability of the method when only parts of the system are fully developed an available or when otherwise only limited design knowledge is available. Besides, the classical dependability concept of fault-error-failure for computing systems in the case of fault trees has the disadvantage that fault trees are an event-based logical construction, where the fault model must be known a priori [5]. This is particularly difficult for software-dominated real-time systems, where legacy code might be integrated on newer platforms that were not intended at the time the code was written, e.g. platforms with a different caching strategy. Thus, effects the code might have cannot be reflected in fault trees even if they were thoroughly maintained during software development. Furthermore, the results of both methods are only valid for exactly one system configuration.

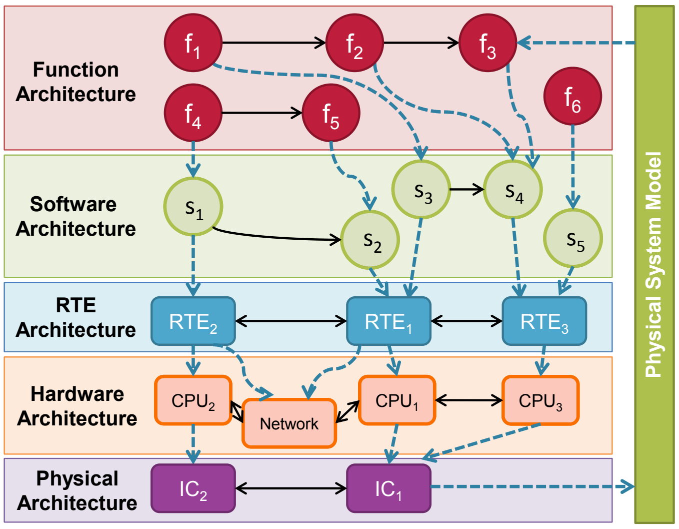


Fig. 1. Example system with a layered system architecture

Another particular challenge in the static approach in FMEA, FTA and current safety standards is that systems are not developed with one holistic model alone. A common way is to use layered architectures, as depicted in Figure 1. The challenge is due to the fact that every change in a layered architecture can possibly interfere with every chain of events in the system, i.e. also on other layers. If there is no systematic way to rule out that other components may be affected, previous results cannot be reused. In conclusion the development process of resilient safety-critical systems suffers from a number of aspects. First, current safety assessment methods are to static to cope with agile or incremental design, e.g. applying an update or adding additional functionality in a second product revision. Second, they cannot deal with the fact that design knowledge is limited in the early phases of system design or even during integration on an execution platform. This is especially obvious if one would try to conduct an FMEA on a single architecture layer alone, e.g. the function architecture, since it cannot capture the effects the execution platform has on its behavior.

II. Contribution

As seen in the example system in Figure 1, a layered architecture does not imply a strict hierarchy between the individual layers. Every layer has its own infrastructure with dependencies relevant to safety analysis. As an example, Run-Time Environments (RTEs) communicate and thus are coupled in a bidirectional manner. Hence, an assessment of all paths implicitly established by the layers is necessary. Furthermore, effects that endanger a safe partitioning for sufficient independence can originate on different layers and can propagate over many layers back and forth. For instance the cause of a timing failure on the functional level might reside in the physical architecture of the platform.

We encourage the use of analysis tools, such as our implementation of a cross-layer dependency analysis framework and to disclose dependency paths and extract them efficiently. To address the issue of hidden dependency paths, we presented in [6] our approach for performing a cross-layer dependency analysis, that is further capable of dealing with the ubiquitous presence of uncertainty in the layer models. Further we demonstrate how the tool can support an incremental design process for robust-systems designs by adding additional design knowledge after each automatic analysis run. We thereby show how a cross-layer dependency analysis can be performed even with limited knowledge and is thus beneficial for safety-critical system design processes, since our conservative modelling approach also allows to reveal hidden dependencies and future design pitfalls that introduce dependencies, which either need careful analysis or might not be resolvable at all.

We see this as a clear advancement over the state of the art, since quite a number of possible dependencies might not be obvious to individual designers or design teams since they only originate in the integration process of a system. For instance a control engineer or function developers might be agnostic about the implications of scheduling on the resource their control algorithm will execute. The use case will also outline how design decisions, w.r.t. partitioning of the system, can be investigated early in the design space exploration phase of a system and what proofs of independence would be necessary to implement the design in this way.

Another result of analyzing a system design in early phases by this approach is that it can explicitly highlight hot-spots that might make the system susceptible to dependency effects, e.g. the propagation of errors or failures, since many elements of the system depend on it. We show how the result of an independence proof, either obtained automatically or explicitly modelled from expert knowledge, can augment the system model an thus be used to eliminate dependency paths in an incremental dependency analysis run.

References

[1] *IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems*, 2nd ed., IEC, April 2010. 1

[2] *ISO 26262 - Road vehicles Functional safety*, International Organization for Standardization - ISO, 2011. 1

[3] IEC, "Fault tree analysis (FTA)," The International Electrotechnical Commission, Tech. Rep. IEC 61025, December 2006. 1

[4] ——, "Analysis techniques for system reliability - procedure for failure mode and effects analysis (FMEA)," The International Electrotechnical Commission, Tech. Rep. IEC 60812, January 2006. 1

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

[6] M. Moestl and R. Ernst, "Cross-Layer Dependency Analysis for Safety-Critical Systems Design," in *Proceedings, ARCS 2015 - The 28th International Conference on Architecture of Computing Systems*, Mar. 2015, pp. 1–7. 2

ErrorPro: Software Tool for Stochastic Error Propagation Analysis

Andrey Morozov, Regina Tuk, Klaus Janschek
Institute of Automation
Faculty of Electrical and Computer Engineering
Technische Universitat Dresden (TU Dresden)
Germany
{andrey.morozov, regina.tuk, klaus.janschek}@tu-dresden.de

I. Extended abstract

Embedded hardware and software systems are vulnerable to environmental impacts, such as single event upsets or electromagnetic interference that may cause silent data corruption and result in data errors at software level. The occurred errors can propagate further through heterogeneous system components. Error propagation has significant influence on the system behavior in critical situations.

Stochastic analysis of this phenomenon gives sound support for fault-tolerant system design, especially in early stages of development. Industrial standards (ISO 26262 and IEC 61508) require to conduct reliability and safety assessment using failure modes and effect analyses (FMEA), hazard and operability studies (HAZOP), fault trees analysis (FTA) or similar methods. The error propagation analysis, as presented in this contribution, provides quantitative estimation of the likelihood of error propagation to hazardous parts of a system, which is an important input for the mentioned methods. It also helps to identify most critical components that should be equipped with error detection or error recovery mechanisms and it assists selecting an appropriate testing strategy, helping to generate such a set of test-cases that will stimulate fault activation in the critical components. Accurate error propagation analysis can be used also for system diagnostics. It helps to trace back an error propagation path up to an error-source that speeds up error isolation, in the case of error detection in observable system outputs.

This article presents a new software tool for comprehensive stochastic analysis of data error propagation through cross-domain systems - ErrorPro. The software tool is based on an extended and improved version of a recently introduced dual-graph error propagation model (DEPM) [1]–[4], an abstract mathematical framework describing structural and behavioral system properties that have the most influence on data error propagation processes. DEPM describe a system as a set of elements. Each element represents an abstract executable part of the system. Two directed graph models are defined using this set: a control flow graph (CFG) and a data flow graph (DFG). A control flow graph represents a possible order of execution of system elements, which play the role of CFG nodes. The arcs of the CFG are weighted with transition probabilities. A data flow graph is a structural representation of a data flow. The DFG of the system contains the same set of nodes as the CFG. Arcs of the DFG show the possibility of data transfer between the elements. They also are considered to be the paths of data error propagation. Faults can be activated in the elements during their execution and result in the occurrence of data errors. Error propagation through the system comprises two aspects: error propagation between the elements and error propagation through the elements. The error propagation between the elements is determined by the DFG structure. The error propagation through the elements depends on the properties of a particular element. Fault activation and error propagation probabilities are defined for each element. Concurrent Markovian analysis of the CFG, the DFG, and these probabilistic properties forms a backbone of DEPM application.

ErrorPro consists of three modules: (i) underlaying error propagation library, (ii) graphic user interface (GUI), and (iii) model transformation algorithms. The underlaying Python source library is a core part. The library allows a user to build a DEPM, to modify it, to specify probabilities of faults activation and errors propagation for single elements, to store the DEPM into an XML-file, and to perform model checking and quantitative analysis. ErrorPro supports an analytical request/response mechanism (see Fig. 1). A user has to formulate an analytical request that defines what has to be computed, e.g. the probability of error propagation to a given set of critical elements, or the mean number of erroneous values in a specified memory slots during a given time interval. For efficient computation, ErrorPro generates discrete time Markov chain (DTMC) models and computes them using a built-in interface with PRISM software [5]. A QT-based GUI is implemented on top of the error propagation library. The GUI visualizes DEPMs and allows a user to interact with them, providing an easy access to all functionality of the library. GUI uses Graphviz [6] for automatic layouting of CFG, DFG, and state graphs of DTMC models. However, the library can be used separately from the GUI in order to embed DEPM functionally into a third-party software. Model transformation algorithms is the third part of ErrorPro. They allow automatic generation of DEPMs from other available baseline system models such as MATLAB Simulink and Stateflow or UML/SysML. At the moment ErrorPro is equipped with a MATLAB parser that transforms Simulink and Stateflow models into DEPMs. Algorithms for DEPM generation from UML Activity Diagrams and SysML Internal Block Diagram are in development as well as a prototype of bidirectional transformation with AltaRica [7].

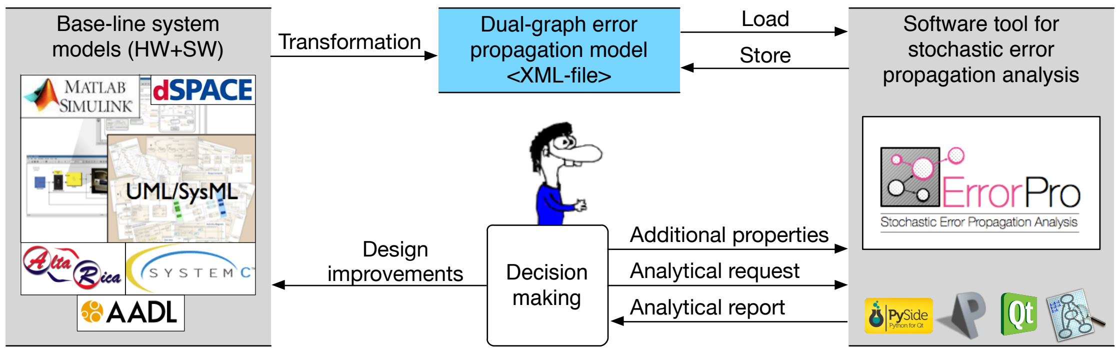


Fig. 1. Application of ErrorPro software for stochastic error propagation analysis.

The article gives detailed information on the DEPM model and algorithms for generation and computation of DTMC models according to user's analytical requests. The article also discusses technical highlights of software design, implementation, and application. The last chapter demonstrates the capabilities of ErrorPro using a real world case study.

References

[1] A. Morozov and K. Janschek, "Dual graph error propagation model for mechatronic system analysis," *In: Proceedings of the 18th IFAC World Congress, August 28 - September 2, Milano, Italy*, pp. 9893–9898, 2011.

[2] A. Morozov and K. Janschek, "Case study results for probabilistic error propagation analysis of a mechatronic system," *In: Tagungsband Fachtagung Mechatronik 2013, Aachen, 06.03.-08.03.2013*, pp. 229–234, 2013.

[3] A. Morozov and K. Janschek, "Probabilistic error propagation model for mechatronic systems. Mechatronics (2014).." http://dx.doi.org/10.1016/j.mechatronics.2014.09.005.

[4] K. Janschek and A. Morozov, "Dependability aspects of model-based systems design for mechatronic systems," pp. 15–22, March 2015.

[5] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)* (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of *LNCS*, pp. 585–591, Springer, 2011.

[6] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *SOFTWARE - PRACTICE AND EXPERIENCE*, vol. 30, no. 11, pp. 1203–1233, 2000.

[7] M. Batteux, T. Prosvirnova, A. Rauzy, and L. Kloul, "The altarica 3.0 project for model-based safety assessment.," in *INDIN*, pp. 741–746, IEEE, 2013.

Fault Injection in Multi-Domain Physical System Models at Different Levels of Abstraction

Raghavendra Koppak[1], Oliver Bringmann[1], Andreas von Schwerin[2]
[1] *University of Tuebingen, Germany*
[2] *Siemens AG, Germany*
raghavendra.koppak@uni-tuebingen.de, oliver.bringmann@uni-tuebingen.de, andreas.schwerin@siemens.com

***Abstract*—Fault injection into simulation models is widely used for validating the satisfactory operation of the safety-critical embedded systems under unexpected conditions like intrinsic failures of electronics or failures caused by the environment. A high level of abstraction of models is required for fast simulation of the systems and fault simulation is a trade-off between accuracy and simulation performance. To capture full system behavior including possible failures, we bring together different domains in one simulation platform. We propose an approach, in which a more accurate physical model is simulated to capture the faults effects and the results are transferred in a second step onto an abstract model for higher simulation performance. Furthermore, we propose an approach to inject faults in physical component models, modeled using Simulink in a virtual platform and, also in a Hardware-in-the-loop (HIL) system. We use an industrial motor control application example to evaluate the methodologies proposed.**

I Introduction

Industrial electronic systems that control manufacturing and machine motion in modern, highly automated production facilities are required to properly cope with failures of all kinds to guarantee safety of operators and machine integrity at any time. Validation of this ability today is still mostly done by final integration and system tests on hardware after development. These tests are complex and expensive already today and are not able to completely cover all possible kinds of failures, as certain failures cannot be provoked in real hardware. Moreover, the late execution of the tests may cause long iteration loops in case weaknesses are detected in the final tests. The validation becomes even more challenging for the highly flexible, self-configuring, self-healing control systems of future manufacturing scenarios with focus on automation of small lot sizes, which requires much higher configurability of the systems and closer interaction between humans and manufacturing machines. A virtual stress test methodology based on fault injection [1] into virtual platforms [2] can ease and speed up system validation for safe operation. In this work, we propose a new approach to fault injection, which supports accurate and still fast simulation of multi-domain simulation platforms.

II Fault Injection in Physical Component Model in a Virtual Platform

Industrial and automotive control systems in general consist of digital electronic components with embedded software (SW) running on one or multiple processor cores as well as analog electronics, mechanical components and the environment. In a virtual hardware-in-the-loop (vHIL) [6] environment, a virtual control unit including processor model and peripherals is connected in closed-loop with a physical system model. We use vHIL setup to bring these multi-domain subsystems of the control system in one simulation platform in a closed approach. Hence it is necessary to bridge the gap between the models of these heterogeneous subsystems and also interaction between them becomes very crucial. Failure effect simulation in such systems can only be done based on virtual prototypes at high abstraction levels, which allow for execution of real SW stacks within only minutes of simulation time.

III Transfer of Faults Effects onto an Abstract Model

The key to capturing physical system behaviour in presence of faults is to have an accurate and appropriate multi-domain simulation model of a physical system, which incorporates important mechanisms such as electrical effects, magnetic effects, mechanical loading etc. Modeling these mechanisms is difficult and more importantly simulating such a system is very slow. We propose to go from an accurate simulation model to the abstract representation which is fast and detailed enough to simulate the behaviour of the overall system. Once we have both the accurate and the abstract simulation models, the effect of a particular fault can be transferred by following these steps. (i) Simulate the accurate model in presence of the fault and capture the behaviour (ii) Transfer the fault behaviour onto the abstract model and simulate the overall system.

IV Implementation of Fault Injection into Physical Component Models for Virtual Platform

Non-digital parts of the system are often modeled in industry using MATLAB/Simulink [7]. The following steps are necessary for failure effect simulation with virtual prototypes of this kind.

IV-A Preparing the Simulink Model to Inject Faults

The principal idea is to extend the Simulink model by adding additional blocks to provide a mechanism to inject fault values. This is achieved by providing a possibility to insert a fault value and also a control mechanism to select either of them. Finally, control and fault signals are externalized as input ports in a top-level system, so that the values can be injected into the system. A new fault injection block (fiblock) is inserted for each line of interest, which simulates the effect of faults. The output of fiblock is equal to the actual value (non-faulty value), when the fault-injection is not activated. A simple example is shown in Figure 1, in which a fiblock is inserted between source and destination Simulink blocks, A and B respectively. For a branched line as shown in Figure 2, a fiblock can be either inserted close to source or destination blocks. In order to avoid an additional fiblock, it is inserted close to Simulink source block A.

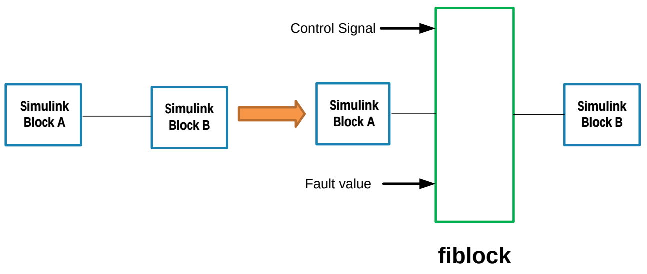


Figure 1: Normal line

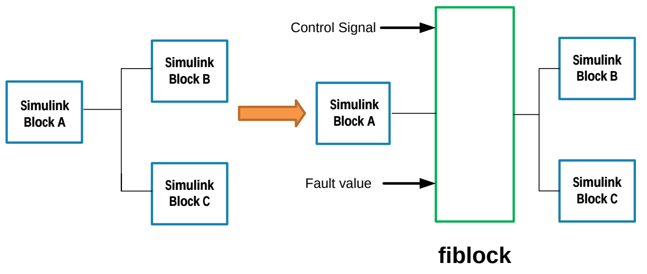


Figure 2: Branched line

IV-B Import and Control Fault Injection in the Adapted Simulink Model Inside a Virtual Platform

For building and simulating virtual prototypes we use the Virtualizer tool suite of Synopsys [8]. Synopsys provides the Virtualizer-Simulink Integration (VSI) library for integrating Simulink models into Virtualizer.

In a first step, the Simulink model has to be adapted by adding appropriate VSI library blocks for the signals which are to be externally controlled in a virtual platform. Identify the Simulink signals that shall be interfaced to the SystemC simulation. Then add connector blocks for these signals to the Simulink system as shown in Figure 4. Then invoke Simulink Coder to generate C++ code for the Simulink system. This step will create a set of files that will be compiled into a library by Simulink. In addition this step will create a SystemC module which contains the interface description of the ports. This SystemC module can be imported into virtual platform without much effort, since all the required scripts and files are automatically generated by Simulink Coder. Figure 3 shows how a Simulink system (Simulink Plant Model) is exported to SystemC. The registers (fifo) between the Simulink system and the connector blocks are added and a library is created with a SystemC interface. The original Simulink Scheduler is responsible for driving the simulation of the Simulink system with the fifos being the boundary, while the SystemC scheduler drives the rest of the simulation. In this setup, Simulink is the slave while the SystemC scheduler acts as the master.

In order to control fault injection during virtual platform simulation, we make use of the SCML command processor. The SystemC Modeling Library (SCML) by Synopsys [8] is a standards-based, TLM-2.0 [9] compatible API library which eases the creation and integration of user-defined SystemC-TLM (Transaction Level Modeling) peripheral models for use in a virtual prototype. The SCML command processor [10] is an SCML object that provides the link between an interactive debugger and the simulation. It allows the debugger to execute commands in the simulation. The command processor operates local to a specific component in a model and can manage multiple commands each with their own set of arguments. In our current investigations, we have used the SCML command processor to control fault injection by defining and implementing the necessary commands. SCML commands are implemented using the following two macros:

- SCML_COMMAND_PROCESSOR macro is used to register which method should be called when the debugger sends a command to the object
- SCML_ADD_COMMAND macro is used to declare the commands that can be handled by the scml_command_processor object

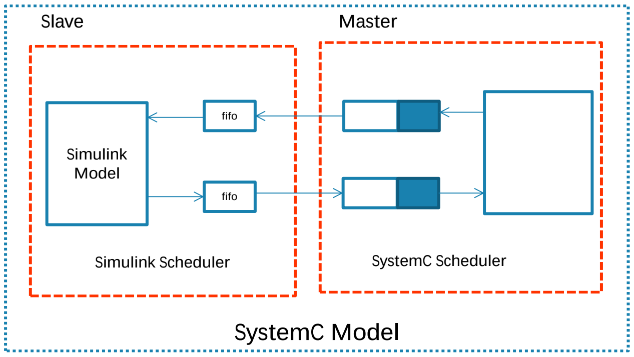


Figure 3: Simulink model import [11]

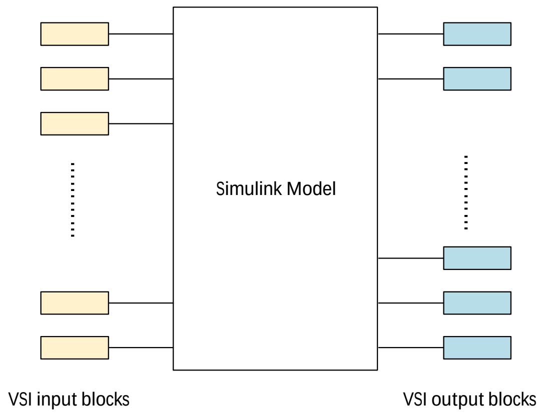


Figure 4: Extended Simulink model

V Fault Injection in Physical Component Models in a HIL System

The final step of validating a newly developed embedded electronic system, in particular in the domains of automotive and industrial motion control, is to use a Hardware-in-the-Loop (HIL) [12] set-up, where the final implementation of the embedded control electronics is connected to a real-time simulation system. Also in such a configuration, fault injection into the physical components that run in the real-time simulation, is needed to validate the proper implementation of previously developed failure reaction mechanisms in final hardware and SW. In case the adapted Simulink model used before in the virtual prototype is abstract enough to be executed in real time, it can be re-used for this task. It has to be imported into the real-time simulation system and the fault injection has to be controlled by proper means. In our investigations, we use the modular dSPACE real time simulation system with DS1006 processor board [13].

Once the simulink model is extended for fault injection as described in IV-A, it is prepared for execution on the dSPACE-system by using the Real-Time Interface (RTI) library provided by dSPACE [14]. The standard Simulink Coder based flow is then used to upload the prepared model to the dSPACE real time simulator, which is connected to the new controller. To control fault injection in such a set-up, dSPACE ControlDesk [14] is used to modify the fault values and control signals.

VI Industrial Motor Control Application

To demonstrate the new methodology an example of industrial drive system as shown in Figure 5 is used. It consists of a Control Unit (CU) and a Power Module (PM) which need to communicate with each other.

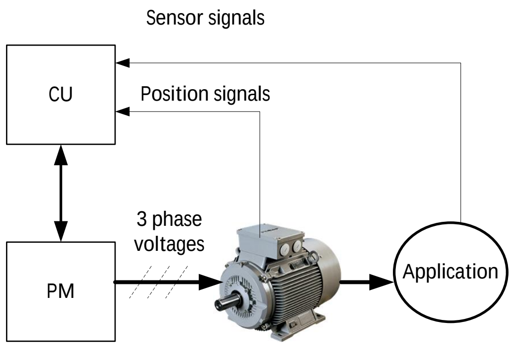


Figure 5: Industrial Drive application

In case of the virtual prototyping approach, the CU instance is represented by a Virtualizer platform built around an ARM fast model [15] processor as shown in Figure 6. It controls the entire application by sending/receiving telegrams as TLM transactions to/from the simulink plant model. The transactors are generic adapter components which decode the information received on their inputs and forwards them. The scml command processor is used to perform fault injection during runtime.

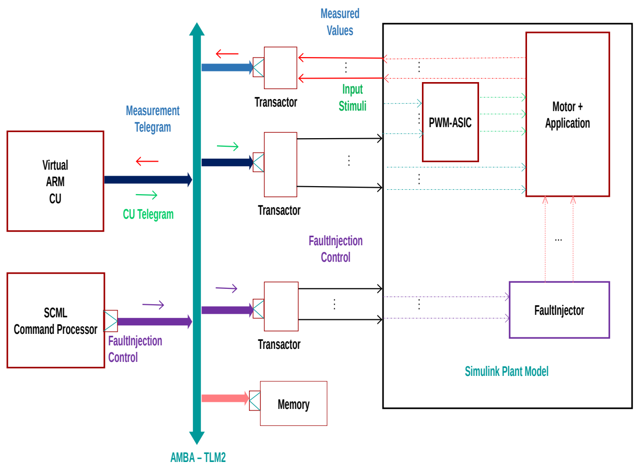


Figure 6: Virtual Platform

In case of the HIL set-up, a real control unit (from the Siemens [16] product portfolio), which controls the entire drive application, is used and connected to the dSPACE system. The PM instance, the motor and the plant application are modeled using Simulink and simulated in the dSPACE real time simulation system.

The three phase induction motor is driven during speed regulation. The speed of the motor is estimated from terminal voltages and currents. The induction motor is fed by a PWM voltage source inverter. The speed control loop uses a PI controller to produce the flux and torque references for the Field Orientated Control (FOC) [17]. The FOC controller computes the three reference motor line currents corresponding to the flux and torque references and then feeds the motor with these currents using a three-phase current regulator. Motor current, speed , torque and fault injection status signals are available at the output. The inputs, the speed set point and the torque set point are also shown.

VI-A Virtual Platform Simulation and Fault Injection Experiments

The simulation output without fault injection is as shown in Figure 7. At time t = 0 s, the speed set point is 500 rpm. As clearly seen, the speed follows precisely the acceleration ramp. At t = 0.5 s, the full load torque is applied to the motor shaft while the motor speed is still ramping to its final value. This forces the electromagnetic torque to increase to the user-defined maximum value (1200 N.m) and then to stabilize at 820 N.m once the speed ramping is completed and the motor has reached 500 rpm. At t = 1.5 s, the speed set point is changed to 1500 rpm and load torque changed from 792 N.m to 0 N.m. The speed increases to 1500 rpm by precisely following the acceleration ramp and it stabilizes at 1500 rpm.

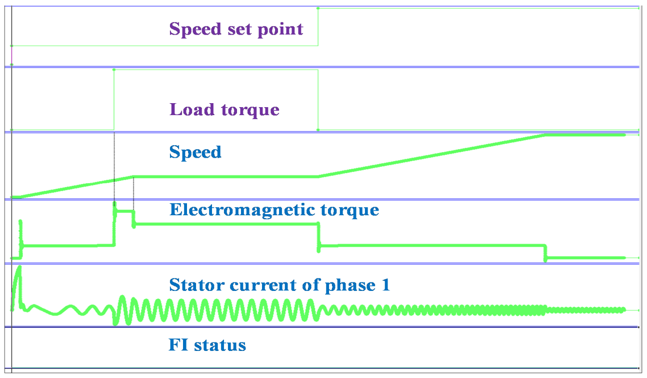


Figure 7: Without fault injection

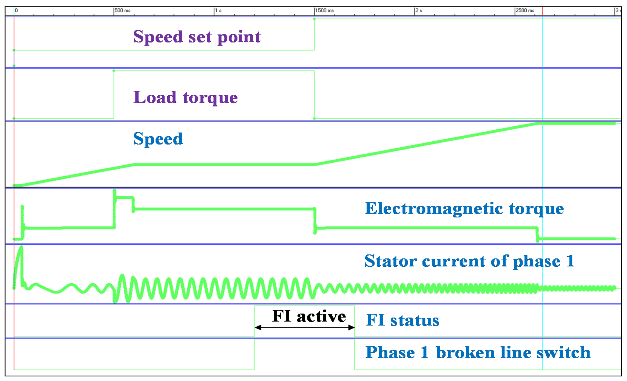


Figure 8: One phase line broken

Figure 8 shows the simulation output in which one of the input phase lines broken fault is activated at time t=1.2s, the motor still continues to run but with reduced output power. Whereas when two input phase lines are broken as shown in Figure 9, there is no rotating magnetic field to produce torque on rotor hence the motor deaccelerates. As soon as the fault injection is de-activated at t=1.7s, the rotor starts rotating and accelerates steadily to reach the speed set point (1500rpm).

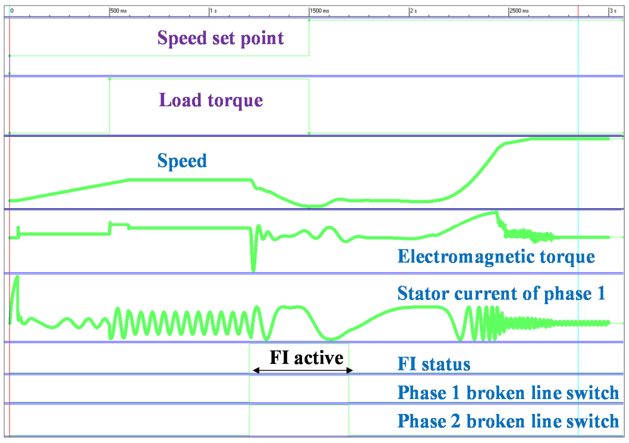


Figure 9: Two phase lines broken

Figure 10 shows the simulation output in which an overload fault is simulated by increasing the load torque more than the maximum load torque of the motor. As a result, the motor starts deaccelerating at t=1.2s and once the load is changed to normal value at t=1.7s, the motor starts accelerating again.

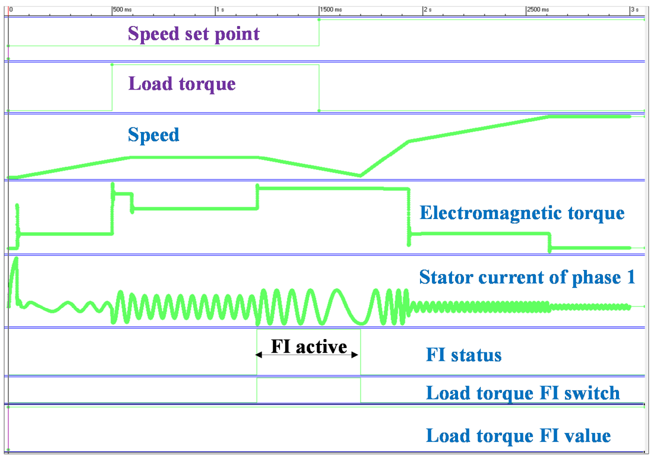


Figure 10: Overload

The Figures 11 and 12 show the simulation output when the short circuit faults on one phase and two phases are simulated respectively.

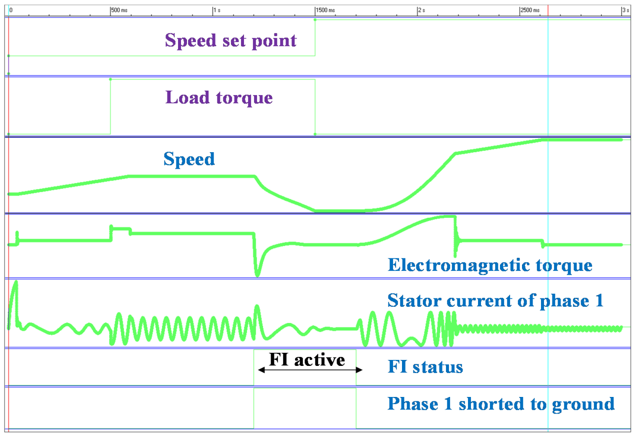


Figure 11: One phase line shorted to ground

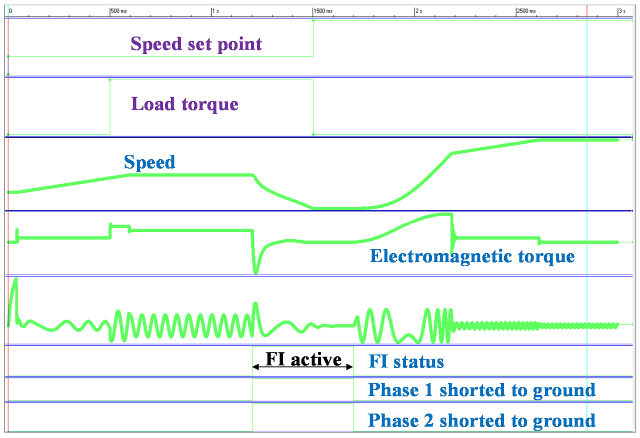


Figure 12: Two phase lines shorted to ground

VII Conclusion and Future Work

It is essential to have software models to accurately match and predict the final design reality. Additionally, these software models should provide a possibility to inject faults. The methodologies proposed in this paper not only help to develop a virtual hardware-in-the-loop (vHIL) system with the possibility to inject faults but also fast enough to simulate. Additionally, the same abstract model with embedded fault injection possibility can be used in the traditional hardware-in-the-loop (HIL) simulation.

With the results of this work, we would like to explore more on the following points in our future work:

- Transfer of fault effects onto an abstract model.
- Fault injection experiments in physical component models in a HIL system.
- Automated fault injection experiments in physical component models in a virtual platform.

VIII Acknowledgement

This work has been partially supported by the German Ministry of Science and Education (BMBF) in the project Effektiv under grant 01IS13022.

References

[1] H. Ziade, R. A. Ayoubi, R. Velazco, and others. A survey on fault injection techniques. Int. Arab J. Inf. Technol., 1(2):171-186, 2004.

[2] Oliver Bringmann, Wolfgang Ecker, Andreas Gerstlauer, Ajay Goyal, Daniel Mueller-Gritschneder, Prasanth Sasidharan, Simranjit Singh. The Next Generation of Virtual Prototyping: Ultra-fast Yet Accurate Simulation of HW/SW Systems. Design, Automation and Test in Europe (DATE) 2015.

[3] Sébastien Tixeuil, William Hoarau , Luis Silva. An Overview of Existing Tools for Fault-Injection and Dependability Benchmarking in Grids.

[4] M. C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. Computer, 30(4):75-82, 1997.

[5] Rickard Svenningsson, Jonny Vinter, Henrik Eriksson, Martin Trngren. MODIFI: A MODel-Implemented Fault Injection Tool

[6] White Paper by Synopsys Virtual Hardware In-the-Loop: Earlier Testing for Automotive Applications

[7] Matlab-Simulink. http://www.mathworks.com.

[8] Synopsys. Virtualizer - Inc. http://www.synopsys.com/

[9] IEEE 1666 SystemC TLM-2.0 standard. http://www.accelera.org/

[10] ”SCML Reference Manual”. Synopsys, Virtualizer Tool Suite Documentation.

[11] ”Third-Party Simulator Integration Manual”.Synopsys, Virtualizer Tool Suite Documentation.

[12] C. Kleijn, Introduction to Hardware-in-the-Loop Simulation

[13] DS1006 Processor Board: http://www.dspace.com

[14] ControlDesk and Real-Time Interface (RTI). http://www.dspace.com

[15] http://www.synopsys.com/Prototyping/VirtualPrototyping/VPModels/Pages/ArmTLMLibrary.aspx

[16] Integrated Drive Systems. http://www.siemens.com

[17] Riccardo Marino,Patrizio Tomei,Cristiano M. Verrelli. Induction Motor Control Design.

Comparison of Different Fault-Injection Methods into TLM Models

Bogdan-Andrei Tabacaru*†, Moomen Chaari*†, Wolfgang Ecker*†,
Thomas Kruse*, and Cristiano Novello*
*Infineon Technologies AG - 85579 Neubiberg, Germany
†Technische Universität München
Firstname.Lastname@infineon.com

Abstract—**The introduction of the ISO-26262 safety standard for automotive applications has increased the verification complexity of safety-critical systems-on-chip (SoCs). Safety verification of SoCs on gate and register-transfer level suffers from long simulation times and exponentially growing fault spaces (i.e., fault location, fault activation time, and fault type). Because of their higher abstraction level and faster simulation speeds, virtual prototypes (VPs) have been introduced to shift the safety-verification process closer to the concept phase and lessen the safety verification effort. TLM has become the *de facto* standard for virtual prototyping; however, TLM lacks built-in fault-injection capabilities. In this paper, we present an approach by which fault injection into TLM-based virtual prototypes is made possible. Our approach represents extensions to the TLM standard sockets, interfaces, and payload and enable abstract fault-model definitions and non-intrusive fault injection during a simulation. The approaches have been applied on TLM models of a RISC architecture.**

Keywords—Fault Injection, TLM, Safety Verification, Virtual Prototyping

I. Introduction

Digital as well as analog and mixed-signal circuits are becoming more complex due to increasing consumer demands on the one hand and due to emerging safety standards such as ISO-26262 [1] on the other hand. As a direct result, the test and verification costs of such hardware models increase exponentially which, finally, lead to higher difficulties to meet time-to-market deadlines.

To address the verification demands introduced by the new safety standards, fault-injection tools and techniques have been developed for hardware models to strengthen the safety-verification process and to optimize the simulation time required for safety-verification regressions, as well. The amount of injectable faults in a hardware model is inversely correlated to the abstraction level of the hardware model (e.g., gate-level, register-transfer level, transaction level models); therefore, although, TLM models (i.e., virtual prototypes) accelerate the process of SoC functional description and lead to early and more accurate SoC structure definition [2], less faults are injectable therein due to the higher abstraction level.

Virtual prototypes (VPs) have been successfully used for early software verification by abstracting the hardware components which the software is developed for. However, compared to their RTL counterparts, VPs still suffer from less tool support and methodology features (e.g., VPs lack simulator support with inherent fault-injection features) which considerably hinders the development and verification of safety-related applications.

In this paper, we present three approaches developed for non-intrusive fault injection into TLM-based VPs. These methodologies independently extend different parts of the TLM library to introduce additional TLM-compliant initiator and target sockets, a generic payload with fault-injection features, and fault injection interfaces. By using the provided sockets and/or payload when developing VPs, fault effects (i.e., errors) are emulated at the outputs of a VP without changing the original model's structure. Moreover, the fault-model decoupling method presented in this paper helps separate the VP's correct functionality from its faulty behavior.

The remainder of this paper is structured as follows. Section II presents other contributions to the topic of fault injection into hardware models, as well as virtual prototypes. Section III describes the general concept of virtual prototypes and their advantages in comparison to classic HDL-based models. Section V states the methodologies presented in this paper to enable fault injection into virtual prototypes. In section IV the fault models relevant for fault injection into virtual prototypes are defined. In section VI we discuss the application of our methodologies through a case study using a RISC-based CPU model. Section VII discusses a series of measurements that assess the performance and usability of our approaches. Last, section VIII contains the paper's summary and conclusions.

II. Related Work

While several fault-injection tools have been developed over the years for gate-level and register-transfer level (RTL), fault-injection mechanisms for transaction-level models (TLM) are still a relatively new topic with limited research.

In [3], a series of fault-injection techniques are presented for SystemC models. Faults are injected into SystemC models by three basic, independent concepts: 1) saboteurs, 2) mutants, and 3) simulator commands. Although no reference is made about TLM fault injection, the presented fault-injection methods are applicable on virtual prototypes (VPs) as well. These concepts are combined in [4], in which SystemC code is statically analyzed to generate an FSM model of its behavior. Next, the FSM model

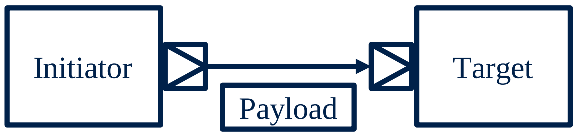


Figure 1. Standard TLM model structure

is used to mutate the original SystemC code and introduce saboteurs in the design. Last, the mutated SystemC models are simulated and fault are activated using control signals in the saboteurs. Thus, an effective fault-injection method with considerable simulation overhead is provided. Furthermore, this concept is not applicable on TLM models because of the TLM-protocol's dependency on dynamic-memory allocation.

An innovative fault-injection method is presented in [5] by using LLVM [6] to create mutated versions of SystemC code. LLVM is used to generate a an intermediate representation (IR) of the original SystemC models. The IR is parsed and a mutated version of SystemC code (e.g., changed operators, changed bit-values, changed event triggers) is generated. This approach does not support mutation of TLM data types (e.g., sockets, generic payload).

ReSP (Reflective Simulation Platform) is a simulation platform implemented in Python [7], [8]. Permanent as well as transient faults are injected into TLM and SystemC models via Python scripts interfaced with the SystemC simulator via `Boost.Python`. However, because private and protected data members are inaccessible with `Boost.Python`, legacy models must be modified. In the case of larger models, the simulation overhead introduced by the Python-based faults modeled increases significantly.

In [9], [10], a non-intrusive fault-injection technique is presented in which C++ virtual-function tables are modified to extend the TLM transport interfaces with fault-injection capabilities. The modification is achieved by mutating the executable TLM models of a LEON3 CPU. To add fault-injection capabilities to the models, this approach augments the TLM initiator and target sockets; however, the approach is platform and compiler dependent. These dependencies are overcome in [11] by applying code mutation on the source code of TLM models. Therefore, faults are introduced directly into the models without manipulation of the C++ virtual-function tables. However, this method is intrusive because of its requirement to change the original code.

A fully non-intrusive fault-injection methodology has been achieved in [12], in which the GNU debugger (GDB) was used to set breakpoints on the transport methods provided by TLM sockets and on generic payload attributes and extensions. GDB's Python API has been used to inject faults into the VPs. However, due to the limited number of hardware breakpoints, the introduced simulation overhead becomes significant.

The fault-injection techniques presented in our paper are focused on fault injection into TLM-based VPs and target the extension of the TLM sockets and generic payload with fault-injection capabilities. By developing VPs with the extended data types, non-intrusive simulation-based fault injection is made possible. Our approach is platform and compiler independent and enables VP fault-model development outside of the TLM model. Our techniques support the injection of multiple faults thanks to the reduced simulation overhead.

III. TLM-Based Virtual Prototyping

Virtual prototyping is the process in which computer software is applied to validate a design before creating a real physical prototype.

In the semiconductor industry, SystemC [13] and TLM [14], [15] have become the *de facto* standards for virtual prototyping. However, while SystemC more closely relates to the HDL-style of modeling and event-based simulation, TLM is an innovative standard which focuses on register accurate modeling, temporal decoupling and a more efficient communication method between models.

The communication process between TLM-based modules (i.e., initiators, targets, interconnects) is achieved by bidirectional objects (i.e., sockets), which transport information through a (generic) payload object (Fig. 1). Abstract communication information is stored in a series of attributes (e.g., command, address, data pointer) in the generic payload and is passed from one TLM module to another by specific method calls (e.g., `b_transport`, `nb_transport_fw`, `nb_transport_bw`).

In this paper, we present three methodologies with which non-intrusive fault injection into TLM models is achieved by changing the information stored in a payload which is passed from one socket to another during a simulation.

IV. Virtual Prototype Fault Models

Gate-level and RTL-specific fault models are not efficiently applicable on VPs because of their bitwise modeling approach (e.g., stuck-at-0, stuck-at-1, bit flip). VPs offer the advantage of abstract fault-model definition (e.g., register address fault, register data fault, interrupt fault, instruction-decode fault) derived from a hardware (HW) model's behavior (i.e., functionality). Therefore, while gate-level and RTL-specific fault models are HW model independent (i.e., reusable on any modeled signal and/or gate), VP fault models are particularly designed for each verified HW model and affect any number of bits from any number of registers. Thus, the specification of such abstract fault models enables system-level safety verification using VPs.

V. Fault-Injection Techniques

The presented approach to inject faults into TLM models possible extends the TLM sockets, interfaces,

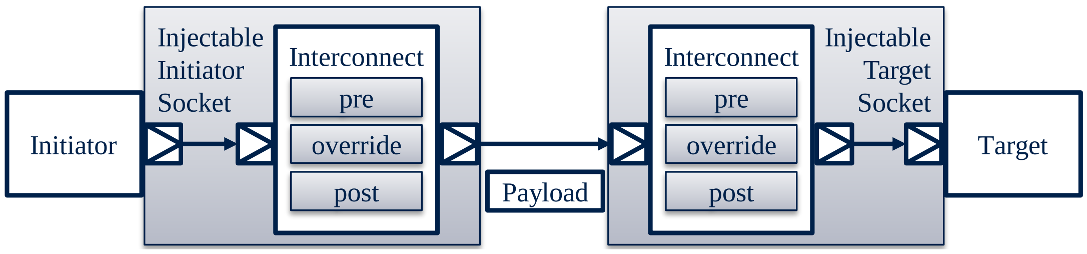


Figure 3. TLM injectable socket block diagram

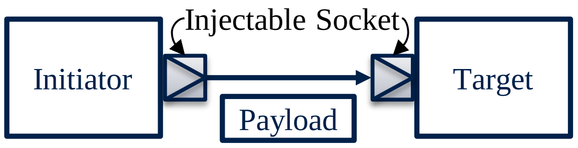


Figure 2. TLM injectable socket

and generic payload and adds extra methods and functionality to the newly available data types. The extended VPs are usable in non-intrusive fault-injection campaigns.

Because fault injection is enabled both in the sockets and at the interface between two TLM models, the propagation of faults between two interconnected TLM blocks is also made available. Therefore, a fault injected into one block is propagated to the next block and becomes an error. By continuous error propagation from block to block, upon reaching the VP's output, the injected fault becomes a failure. Based on this fault-propagation analysis the robustness of a VP is evaluated.

A. *TLM Injectable Socket*

To provide TLM-based VPs with fault-injection capabilities, interconnects have been embedded into the standard TLM initiator and target sockets to sabotage the transport methods' normal functionality (e.g., `b_transport`, `nb_transport_fw`, `nb_transport_bw`) (Fig. 2). To add more flexibility to the standard TLM transport flow, three hooks are provided inside the saboteur-like transport methods of the interconnects, which do not consume simulation time. Fault injection is enabled by connecting a fault injector to one of the available hooks: 1) pre-hook, 2) override-hook, or 3) post-hook (Fig. 3).

The pre-hook gets called before the actual transport method, while the post-hook is called after the transport method is executed. The override-hook replaces the target socket's transport implementation completely and acts like an external (non-intrusive) mutant.

The fault injector is a user-defined class derived from a TLM interface with specialized transport method through which a payload's contents are changed during the simulation.

By modeling the VP faults inside the fault injector and connecting the injector to a specific TLM block, fault injection is performed without changing the TLM block's original code.

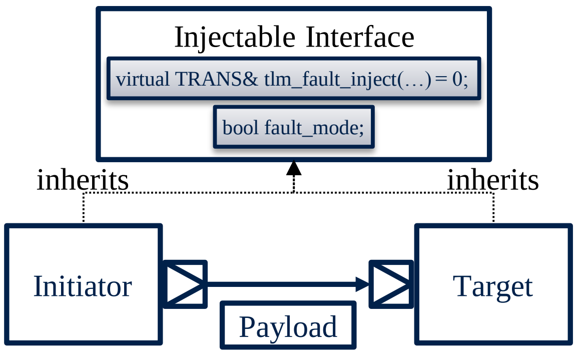


Figure 4. TLM injectable interface

B. *TLM Injectable Interface*

The injectable interface derives the original TLM backward/forward interface class to address fault-injection requirements (Fig. 4). Beside the normal virtual functions defining the blocking and non-blocking interfaces, a configuration data member (i.e., `fault_mode`) is used for the activation of injectable faults. The data member is configurable to set the system in normal or in fault-injection mode. In normal mode, the system continues its ordinary functions calling the TLM interface routines. Once the system goes into fault-injection mode, the `tlm_fault_inject` method is called to inject faults in the system models by corrupting the payload's attributes.

This injection mechanism is usable in the early system-development phase, in which the communication protocol between two or more TLM modules is not yet fully defined. Therefore, this mechanism emulates fault effects on the communication path between the corresponding models. Moreover, the communication protocol's safety requirements can be verified via fault injection before implementing the actual hardware models. As soon as the communication protocol is defined and is included in the VP, the TLM injectable socket defined in section V-A can be efficiently used to inject faults into the communication module as well.

C. *Lightweight TLM Injectable Socket*

The lightweight TLM injectable socket methodology is a variation of approach in subsection

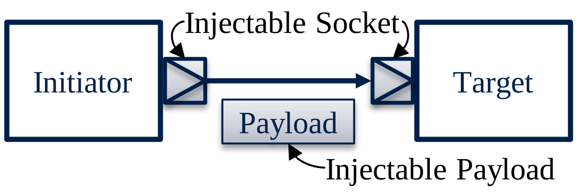


Figure 5. TLM injectable socket and generic payload

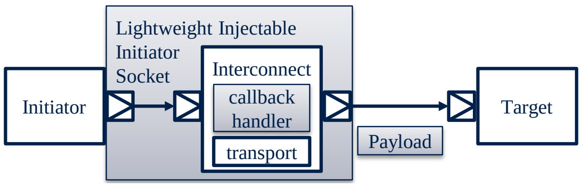


Figure 6. Lightweight TLM injection socket block diagram

V-A, which adopts a lightweight injectable initiator socket and uses an extended generic payload with dedicated fault-injection methods for each generic payload attribute (e.g., `force_address`, `freeze_address`, `release_address`) (Fig. 5).

The injectable initiator socket contains only one saboteur-like interconnect component (i.e., for the initiator socket) as opposed to methodology V-A (Fig. 6), which contains one for the target socket as well. All blocking and non-blocking transport methods are wrapped by the interconnect and are, furthermore, controlled by callback handlers provided for the component's transport methods. The callbacks are used to modify the generic payload's attributes. Using SystemC threads to synchronize the TLM models with injection events accordingly, faults are activated by connecting a callback and deactivated by disconnecting the callback during a simulation.

VP fault models are developed independently from the fault-free TLM module as part of the fault injectors (i.e., callbacks) which are connected to the TLM modules. Therefore, the fault models are effectively decoupled from the VP (i.e., described outside of the model) which they actually affect and become reusable in other VPs by extending the corresponding test case (Fig. 7).

VI. Case Study

The three methodologies presented above have been applied on a RISC-based CPU, NanoRISC, modeled according to TLM's loosely-timed (LT) (Fig. 8) and approximately-timed (AT) (Fig. 9) coding styles.

NanoRISC supports an integer instruction set,

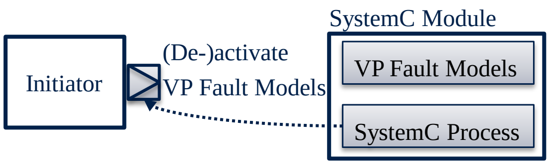


Figure 7. Virtual prototype fault-model decoupling

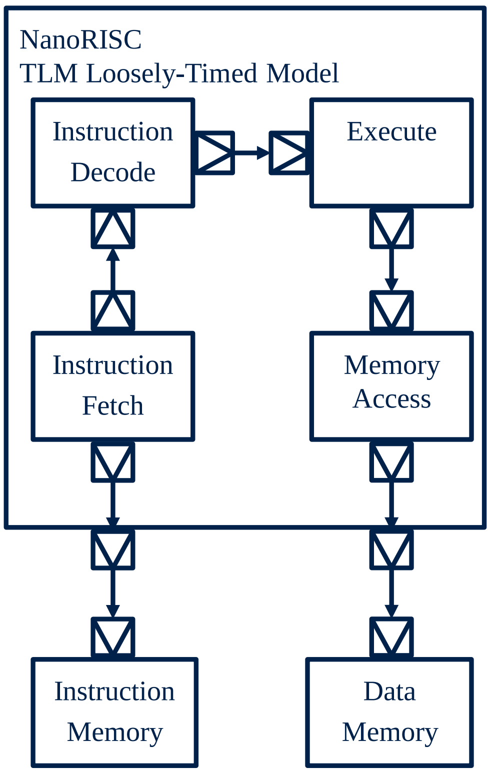


Figure 8. NanoRISC TLM loosely-timed block diagram

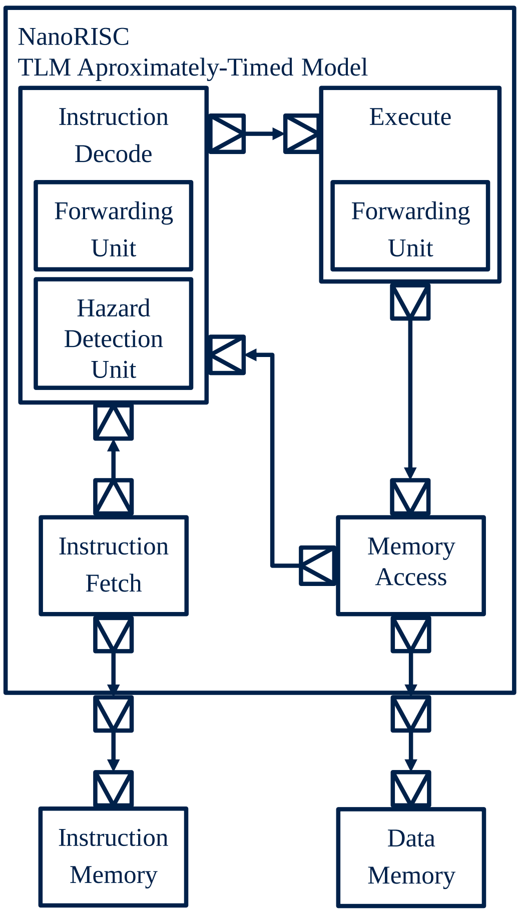


Figure 9. NanoRISC TLM approximately-timed block diagram

Table I. MEASURED MEAN SIMULATION OVERHEAD IN MILLISECONDS (MS)

A	B	C	D	E
Without fault-injection tool	With fault-injection tool, no faults injected	Slowdown factor (B/A)	With fault-injection tool and faults injected	Slowdown factor (D/A)
248.538	253.706	1.0208	258.934	1.0418

Table II. FAULT-MODELING COMPLEXITY AND MEAN SIMULATION OVERHEAD PER INJECTED FAULT

Location of injected fault	Lines of code for fault modeling		Number of faults injected		Number of failures observed		Mean simulation time (ms)	
	TLM LT	TLM AT	TLM LT	TLM AT	TLM LT	TLM AT	TLM LT	TLM AT
RD register	42	51	10	10	7	5	262.368	121.554
Instruction register	23	28	10	10	10	10	253.333	125.348
ALU result register	36	27	5	5	5	5	261.101	153.154

a five-stage pipeline (i.e., instruction fetch, instruction decode, instruction execute, memory access, and write-back), and a hazard-control unit for the introduction of stalls in the CPU. The execution unit is composed of shifting blocks, adders, multipliers, and other logic and arithmetic units.

Each of NanoRISC's five pipeline-stage components has been modeled as an independent initiator (e.g., Instruction Fetch), interconnect (e.g., Instruction Decode), or target (e.g., Instruction/Data Memory). The components communicate with each other via sockets by passing a payload with information regarding addressed registers and solved hazards.

The hazard control unit and the forwarding units are not present in the TLM LT coding style because each instruction is executed sequentially and, therefore, no hazardous behavior is possible. Moreover, although not strictly necessary, the explicit modeling of each pipeline stage comes as an advantage to more easily demonstrate the fault-injection processes provided by our methodologies.

To enhance the TLM-based VPs with our proposed fault-injection techniques, the corresponding data types (e.g., initiator/target socket, generic payload, backward/forward interfaces) have been replaced with their injectable counterparts accordingly. Next, VP fault models have been defined respecting the model's coding style. During a simulation, the fault models have been (de-)activated to inject faults in the VPs.

VII. RESULTS AND DISCUSSION

The performance penalties introduced by the three presented approaches during fault-injection campaigns have proved to be negligible in comparison to simulating the same models without fault injection. This is an expected result because the extra code introduced by the TLM-library extensions and the code required to model the faults and their behavior mainly introduce a slight compilation overhead. Moreover, the negligible simulation overhead is due to the way in which each fault-injection approach has been implemented. On the one hand, all callback connections and disconnections are C++ method calls which do not consume simulation time. On the other hand, the injectable interface operates via the TLM blocking interface and requires no extra synchronization with the models. The main contributors to the simulation overhead are, for example, two SystemC processes required to activate and deactivate the faults during a simulation. Moreover, these processes are only triggered once by the simulator during the whole simulation and are afterwards killed to optimize the simulation. Table I illustrates the measured mean simulation overhead in milliseconds.

To assess the fault-modeling complexity, the number of user-written code lines have been measured for each modeled and injected fault and are presented in Table II. The results are presented for the loosely timed (LT) as well as for the approximately-timed (AT) TLM models of the NanoRISC CPU. On average, the TLM AT models required more code lines to accurately model each fault's behavior because of the more detailed TLM AT protocol. One exception has been observed in the ALU result register fault and was caused by the TLM AT protocol's relatively straightforward implementation. Moreover, the number of injected faults (i.e., the number of executed simulations per modeled fault) as well as the number of simulations which led to system failures are also shown in table II. Apart from the faults injected into the Rd register, all injected faults have successfully caused failures in the system. These results directly illustrate the necessity of extra safety mechanisms to protect the vulnerable registers. Finally, the last two columns in the table present the mean simulation time (i.e., wall time) required to perform all fault-injection simulations and observe whether the injected faults led to failures or were detected and/or corrected during the process of propagating through the system.

The individual fault-model connection methods required by each of the presented approaches (e.g., overloaded payload transactions, callbacks) have not been taken into consideration in the measurement of the fault-modeling complexity. This is because each approach required on average only 8-10 lines of code which were ultimately reused for the simulation of each individually modeled fault.

VIII. Summary

In this paper, three generic fault-injection approaches have been presented to enable safety verification of TLM-models. These techniques have been applied on TLM loosely and approximately-timed models of a RISC-architecture CPU to illustrate the process of extending a TLM model for the purpose of fault injection therein.

By using the presented techniques, faults are injected during a simulation by defining fault models separately from the original TLM models, hence performing non-intrusive fault injection. Moreover, it has been shown that the performance penalties of applying the presented approaches are negligible.

Acknowledgment

This work is partially supported by the German Federal Ministry of Education and Research (BMBF) in the project EffektiV (contract no. 01IS13022).

References

[1] ISO, CD, "26262, Road vehicles–Functional safety," *International Standard ISO/FDIS*, vol. 26262, 2011.

[2] J.-H. Oetjens, O. Bringmann, M. Chaari, W. Ecker, B.-A. Tabacaru *et al.*, "Safety evaluation of automotive electronics using virtual prototypes: State of the art and research challenges," in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. IEEE, 2014, pp. 1–6.

[3] S. Misera, H. T. Vierhaus, and A. Sieber, "Fault injection techniques and their accelerated simulation in systemc," in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*. IEEE, 2007, pp. 587–595.

[4] F. Bruschi, F. Ferrandi, and D. Sciuto, "A framework for the functional verification of systemc models," *International Journal of Parallel Programming*, vol. 33, no. 6, pp. 667–695, 2005.

[5] M. Sousa and A. Sen, "Generation of tlm testbenches using mutation testing," in *Proceedings of the eighth IEEE/ ACM/I-FIP international conference on hardware/software codesign and system synthesis*. ACM, 2012, pp. 323–332.

[6] C. Lattner and V. Adve, "Llvm language reference manual," 2006.

[7] C. Bolchini, A. Miele, and D. Sciuto, "Fault models and injection strategies in systemc specifications," in *Digital System Design Architectures, Methods and Tools, 2008. DSD'08. 11th EUROMICRO Conference on*. IEEE, 2008, pp. 88–95.

[8] G. Beltrame, L. Fossati, and D. Sciuto, "Resp: a nonintrusive transaction-level reflective mpsoc simulation platform for design space exploration," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 12, pp. 1857–1869, 2009.

[9] A. d. Silva and S. Sanchez, "Leon3 vip: a virtual platform with fault injection capabilities," in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*. IEEE, 2010, pp. 813–816.

[10] A. d. Silva, P. Parra, Ó. R. Polo, and S. Sánchez, "Runtime instrumentation of systemc/tlm2 interfaces for fault tolerance requirements verification in software cosimulation," *Modelling and Simulation in Engineering*, vol. 2014, p. 42, 2014.

[11] V. Guarnieri, N. Bombieri, G. Pravadelli, F. Fummi, and H. e. a. Hantson, "Mutation analysis for systemc designs at tlm," in *Test Workshop (LATW), 2011 12th Latin American*, March 2011, pp. 1–6.

[12] B.-A. Tabacaru, M. Chaari, W. Ecker, and T. Kruse, "A meta-modeling-based approach for automatic generation of fault-injection processes," *DVCon Europe*, pp. 1–7, 2014.

[13] G. Arnout, "Systemc standard," in *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*. ACM, 2000, pp. 573–578.

[14] O. S. Initiative, "Transaction level modeling (tlm) library," 2008.

[15] T. De Schutter, *Better Softwarer. Faster! Best Practices in Virtual Prototyping*. Synopsys Press, 2014.