

A Rapid Prototyping System for Error-Resilient Multi-Processor Systems-on-Chip

Matthias May, Norbert Wehn
Microelectronic Systems Design
Research Group
University of Kaiserslautern
67663 Kaiserslautern, Germany
{may, wehn}@eit.uni-kl.de

Abdelmajid Bouajila, Johannes Zeppenfeld,
Walter Stechele, Andreas Herkersdorf
Institute for Integrated Systems
Technische Universität München
Arcisstr 21, 80290 München, Germany
{a.bouajila, zeppenfe}@tum.de

Daniel Ziener, Jürgen Teich
Hardware/Software Co-Design
Department of Computer Science
University of Erlangen-Nuremberg
91058 Erlangen, Germany
{daniel.ziener, teich}@cs.fau.de

Abstract—Static and dynamic variations, which have negative impact on the reliability of microelectronic systems, increase with smaller CMOS technology. Thus, further downscaling is only profitable if the costs in terms of area, energy and delay for reliability keep within limits. Therefore, the traditional worst case design methodology will become infeasible. Future architectures have to be error resilient, i.e., the hardware architecture has to tolerate autonomously transient errors.

In this paper, we present an FPGA based rapid prototyping system for multi-processor systems-on-chip composed of autonomous hardware units for error-resilient processing and interconnect. This platform allows the fast architectural exploration of various error protection techniques under different failure rates on the microarchitectural level while keeping track of the system behavior. We demonstrate its applicability on a concrete wireless communication system.

I. INTRODUCTION

With the continuing downscaling of CMOS technologies, static and dynamic variations, time dependent device degradation, sporadic timing errors, and radiation induced soft errors will result in unreliable components. The traditional worst case design methodology becomes infeasible due to the large area and energy overhead and the required a priori knowledge of all error sources at design time. Further technology downscaling is only profitable if the costs for reliability keep within limits. A promising approach is to tolerate errors on the physical hardware level and correct them on higher levels [1]–[4].

In this paper, we focus on the design of multi-processor systems-on-chip (MPSoC) which can tolerate a specific amount of transient errors. Soft errors, one type of these errors, are caused by radiation. When a particle hits a semiconductor device, it can generate an electron-hole pair. This yields a false signal value for a short time, which is called single-event transient (SET). When a particle hit causes a flipped bit in a register, a single-event upset (SEU) occurs. Another type of transient errors are timing errors caused by temporary delay variations.

We present an FPGA based rapid prototyping system for an MPSoC consisting of autonomous hardware units. These units

can monitor and analyze sporadic disturbances and trigger autonomously adequate reactions. Different techniques are integrated into the MPSoC for a holistic protection of the system: a self-correcting data path and control flow checking in LEON3 [5] processor cores and a run-time configurable data protection of the AMBA [6] advanced high-performance bus (AHB).

The system is dedicated for fast architectural exploration under different scenarios for transient errors on the physical hardware level. Errors can be injected on multiple places in the hardware for evaluation. Our adaptive FPGA prototyping system allows a fast design space exploration by emulating various error protection techniques with varying failure rates on the microarchitectural level. The impact on the system behavior can be evaluated with respect to overhead in area and latency. Another alternative for such an exploration is simulation. Since errors occur on microarchitectural level, simulation has to be performed on this level which results in extreme long simulation times if the system behavior has to be monitored over a long time period. This makes simulation infeasible.

A channel decoding system is used as application for demonstration. Channel decoding is an important building block in any communication system. Further, it is representative for probabilistic and iterative algorithms which have an algorithmic resilience. Many other applications also have a cognitive resilience, e.g., video or audio compression. This algorithmic or cognitive resilience, called application level resilience in the following, offers a large optimization potential for error resilient system architectures [7]–[9]. The system designer can make a trade-off between quality and robustness.

The paper is structured as follows. Related work is discussed in Section II. Sections III, IV and V focus on various techniques to detect and correct errors in data path, control path, and interconnect, respectively. The rapid prototyping platform and results from an exploration scenario are presented in Section VI.

II. RELATED WORK

Error detection and correction methods have important roots in the area of fault-tolerance. Here, the most familiar

The presented work was partly done under the scope of the AIS project which is supported by the German Federal Ministry of Education and Research, funding label 01M3083.

methods for error detection or correction are the duplication or triplication, respectively, of a given processor core with subsequent comparison of the results [10]. Duplication or even triplication of processing units is, however, most of the times prohibitive due to area and power consumption. Hence, these approaches are only affordable in safety-critical systems with a very high demand on reliability.

A technique called virtualization-assisted concurrent, autonomous self-test (VAST) [11] uses a concept similar to ours with autonomous, on-line failure protection. VAST tests the processor cores of a multi-core system while they are free from executing tasks. Only hard failures are detected because the on-line test is not running during normal operation.

Related work on data path protection includes arithmetic and logic units protected with residue or parity codes [12]. Ernest et al. [13] protects a CPU pipeline with razor which is a technique for timing error detection and correction. Whenever an error is detected it is corrected by inserting one cycle delay for correction. This technique does not protect the pipeline against SET and SEU errors.

Gaisler [14] developed a fault-tolerant version of the Leon processor. Fault-tolerance is provided against bit-flips (Single Event Upsets - SEU) in cache memories, register file and pipeline registers. This work does not address the increasing problem of timing and single event transients.

Related work on control flow checking can be divided into approaches using an additional hardware checker unit or a watchdog processor [15]–[20], and approaches which are completely software-based [21]. In these approaches, the program code is first structured into basic blocks¹.

Control flow checking using assertions (CCA) [21] denotes a software-based approach. After creating a basic block graph, a sequence of special control instructions is inserted into the program code at the beginning as well as at the end of each basic block. These additional instructions verify that only legal branch or jump destinations according to the specification, given by the basic block graph is taken. A good overview over software methods for control flow checking for security and fault tolerance is given in [22].

To check all types of instructions, a signature (hash or a CRC value) of all instructions of a basic block can be calculated offline (at compile-time). At run-time, a hardware checker can calculate the signature of the executed instruction in a basic block. When leaving a basic block, the signatures can be compared and errors inside the basic block can be found. Signature methods can be divided into two groups, namely embedded signature monitoring (ESM) [15]–[17] and autonomous signature monitoring (ASM) [18], [19].

For interconnect protection, interconnect noise can be reduced [23] or general protection techniques on the circuit level to detect timing errors [13] or to mask SET errors [24] can be applied. On higher levels, spatial- and time

¹A basic block is a sequence of code which is executed successively without any jumps or branches except, possibly, at the end. The basic block can only be left at the end of a block and can only be entered at the beginning. Only the last instruction can be a jump or branch and only the first instruction can be a jump or branch destination.

redundancy can be added [4]. Well known techniques are error detection codes with automatic retransmission request (ARQ) or forward error correction (FEC) codes [25]–[27]. However, the general application of these techniques implies a large overhead in area, energy and timing. In [25] it was shown that the efficiency of the error protection scheme strongly depends on the application constraints.

Bertozzi et al. [28] evaluated AMBA data bus protection schemes using Hamming codes. They conclude that using the Hamming code only for error detection and ARQ is the most effective coding scheme with respect to energy per useful bit. However, retransmission is not feasible for applications with strong latency constraints.

III. DATA PATH PROTECTION

Data path error detection uses Nicolaidis shadow registers [4] which detect SET, SEU, and timing errors. An extra shadow register is added to each inter-stage pipeline register, see Figure 1. Whenever an error is detected the errant operation has to be retried. Assuming that a SET hits the execution stage (EX), it will be detected at the EX/ME inter-stage pipeline registers. The operation should be retried but at that moment the execution stage input registers have already been overwritten. Therefore a straight-forward solution would be to flush the pipeline and to restart it at the errant instruction.

In order to minimize the error recovery overhead, a new customized micro-rollback [29] is presented. As the shadow register technique has an error detection latency of only one clock cycle, storing the last state in history registers (figure 1) is sufficient to perform a micro-rollback [30]. Whenever an error is detected, operations will be retried from the history registers. The recovery penalty equals two clock cycles, one cycle for error detection and one for correction. This error penalty is independent from where the error occurs.

This concept has been implemented in a Leon3 CPU pipeline. Micro-architectural extensions are added to each pipeline register as shown in Figure 1. Whenever an error is detected, the global error signal (obtained by OR'ing all protected registers error signals) is forwarded to the control unit which is extended to control the pipeline rollback. Errors are detected and corrected with a 2-clock cycle penalty.

Processor protection with shadow registers has been implemented in ASIC [13]. However implementing shadow register based designs in FPGAs can not be done in a straight-forward manner as FPGA tools do not allow to constrain hold time, also routing two different clocks to the same clock domain is not simple/impossible with existing tools. To circumvent this problem, we decided to use the flip-flops clock enable signal to mimic the 2-clock scheme of the main/shadow register concept.

IV. CONTROL PATH PROTECTION

Control path protection can be achieved by checking the control flow of a program under execution, and in case of an error a re-execution of the erroneous instruction. A quite general definition of control flow checking may be given as follows: Control flow checking denotes the task to test whether

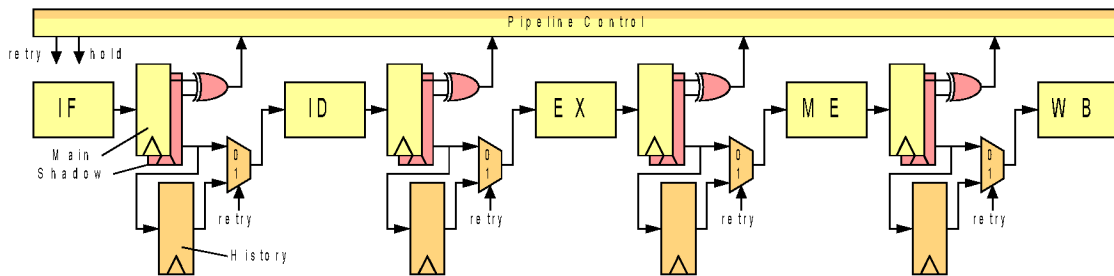


Fig. 1. CPU pipeline protected with shadow registers

a sequence of program counter values is correct with respect to a given program specification.

In SoCs, a CPU often executes only a few specified programs over its lifetime. So, it is beneficial to analyze these subroutines for control flow instructions statically.

Control flow instructions (CFI) can be categorized into conditional branches and unconditional jumps. Conditional branches depend on the result of a logical or arithmetic operation. Both groups of control flow instructions can be further subdivided into direct (static) and indirect (dynamic) jumps or branches. The destination of direct branches or jumps is fixed at compile-time and is encoded into the jump or branch instruction in an absolute or relative address. For indirect jumps or branches, the destination address is determined during program execution.

If we may assume that only direct jumps and branches exist in a given code segment, we are able to check the control flow of this code by verifying the correct execution of each direct control flow instruction as well as the (successively) linear execution of all the other instructions (the program counter value is incremented by one word address after each instruction).

To check the correct execution of control flow instructions, we need to check the correct address of the control flow instruction and the correct target address. The program counter value before and after the execution of a control flow instruction can be compared to these addresses. If there is a mismatch, an error signal may be raised.

Because the start and target address of direct branches and jumps is known at compile-time, it is possible to extract this information from the binary or the disassembled program code by decoding the instructions [31], [32]. With this information, the *control flow instruction graph* (CFIG) can be built. The information of the CFIG can be used as a specification of a control flow checker unit and implemented either directly by a FSM or by micro-instructions of a micro-programmed circuit.

A hardware architecture for control path protection to monitor and to correct the executed control flow instructions of the given Leon3 RISC CPU is introduced also in [31], [32]. This approach can monitor direct jumps and branches as well as call and returns from subroutine. To achieve a correction of a corrupt program, a detected incorrect jump or branch can be re-executed.

The approach consists of an additional checker unit and some few modifications of the Leon3 integer pipeline. The checker unit must know the instruction's program address and

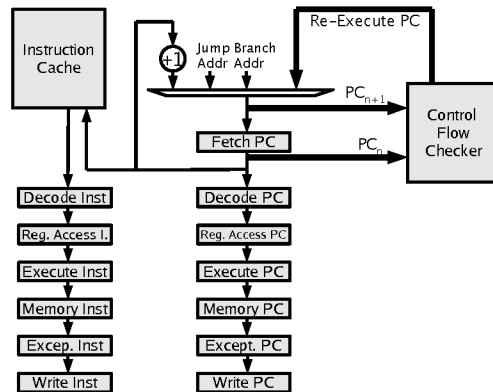


Fig. 2. A checker unit is placed in between the first pipeline stages of the Leon3 core [5] for checking the correctness of branches and jumps. All bold lines denote new signal paths needed for monitoring and re-execution of jump and branch instructions.

the address of the next instruction to execute. We achieve these values from the first stages of the Leon3 integer pipeline (see Figure 2).

Direct jumps and branches are checked with the information of the CFIG stored in an on-chip memory. The correct start and target address of the direct jump or branch are compared with the real values from the processor pipeline.

Returns can be verified in this approach by introducing an additional *hardware stack*. Upon a call (direct or indirect), the return address is stored in the stack. Once the return instruction is executed, the correctness of the target address can be verified.

With this technique, we are able to detect all faults and errors which affect the control flow of the CPU. This includes temporal and permanent faults, appearing in the program memory, cache, program counter generation, as well as communication structures between them.

If an error is detected, the re-execution procedure will be initiated. In this case, the address of the erroneous instruction is forwarded to the fetch stage of the Leon3 integer pipeline unit. The result is a re-fetch of the instruction which causes the error (see Figure 2).

V. INTERCONNECT PROTECTION

Interconnect on the platform is performed via an AMBA AHB [6] bus. This bus is extended by error detection and correction (EDC) units for the data signals. These EDC units are run-time-reconfigurable by the applications running on the Leon3 processors. The applications can set the error protection

TABLE I
AHB DATA BUS PROTECTION MODES AND POSSIBLE DATA TYPES

Mode	Error Protection	Payload	Possible data type
0	parity bit, ARQ	32 bit	address
1	no protection	32 bit	audio streaming
2	3× repetition, voting	11 bit	Turbo decoder, internal
3	sign bit 3×, voting	4×6 bit	parallel decoder input
4..15	extendable with any application specific error protection mode		

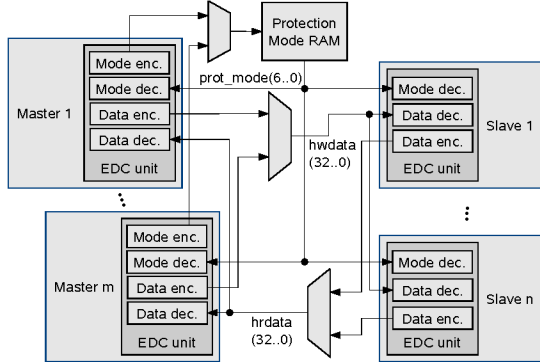


Fig. 3. AMBA AHB data bus error protection

mode appropriate to the type of information sent over the bus. This allows efficiently exploiting an application-level resilience.

Various techniques with simple error detection and correction codes, e.g., parity bits with ARQ, repetition codes with voting, and Hamming codes are implemented for soft error protection. Timing errors on single or multiple bit lines are also detected by these codes. However, it can happen that all bits are delayed by the same amount of time. To detect also these errors, we implemented a phase shifting technique [9], [27]. One data bit is synchronously flipped every second clock cycle at the master and the slave EDC unit.

The AHB protection system is shown in Figure 3. The error protection on the read/write data bus depends on the address of the transfer. The virtual memory is partitioned into segments and the protection mode can be set by the application for each memory segment independently by storing it in the protection mode RAM. The mode identifiers are protected by a (4,7) Hamming code.

Up to sixteen different general or application specific error protection modes can be implemented. An example configuration is shown in Table I.

Some errors can result in a complete system failure and require a hardware reset. To prevent these failures, some special modes for system evaluation with injected errors are implemented. In these modes the non-detected errors are only counted but not sent to the receiving unit.

The retransmission of a data word in the case of an ARQ takes only one additional clock cycle due to the pipelined design of the AMBA AHB. All other currently implemented forward error correction codes imply no additional delay.

VI. RAPID PROTOTYPING PLATFORM

The prototyping system is implemented on a Xilinx Virtex-4 LX100 FPGA [33] on a Gaisler GR-CPCI-XC4V development

TABLE II
AREA OVERHEADS FOR THE CONTROL (CP) AND DATA PATH (DP) PROTECTION OF THE LEON3 CPU. THE PERCENTAGE INCREASE COMPARES THE PROTECTED WITH THE UNPROTECTED CPU.

	Leon3	CP prot	inc. in %	Leon3 + hold regs	DP prot	inc. in %
LUTs	13554	729	+5%	13805	1881	+14%
FFs	3476	202	+6%	5411	766	+14%
Slices	7332	423	+6%	9224	1245	+13%

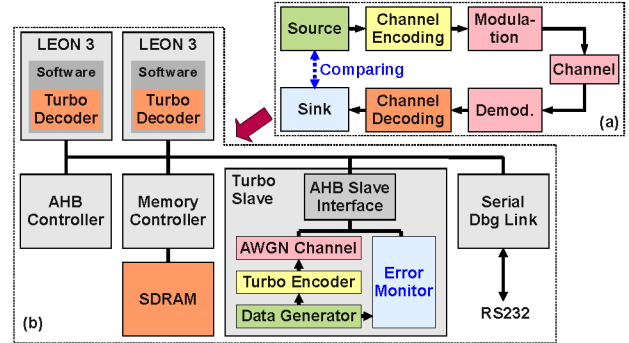


Fig. 4. Channel coding centric communication system (a) mapped on the prototyping platform (b)

board [5] containing multiple Leon3 processor cores [5], an SDRAM controller, and several peripheral units connected via an AMBA AHB. The individual components are protected by the techniques described in the previous sections.

Currently we use a two-processor-system. The implementation of the data bus protection results in a hardware overhead of one BRAM [33] for the protection modes and about 2% more logic cells for the additional bus lines and EDC units. Table II shows the additional area overhead for the data and control path protection, respectively. For monitoring the control flow, we further need 28880 bits of additional memory to store the CFG which corresponds to three additional BRAMs. To prototype the shadow register technique, we use flip-flop clock enable and also add hold registers to keep coherence in pipelined update of main and shadow registers. Therefore, the unprotected Leon3 with hold registers is compared towards the data path protected Leon3 with hold registers (see Table II). Please notice that Leon3 VHDL code does not explicitly separate data and control path, so we did this according to our code understanding.

We demonstrate the rapid prototyping system on a 3GPP LTE [34] Turbo decoding system implemented on the two processors. The outstanding forward error correction of Turbo codes [35] made them part of many today's wireless communications standards, e.g., WiMAX, CDMA2000, UMTS, and 3GPP LTE. Examples for multi-processor implementations of Turbo decoding are [36], [37].

A Turbo decoder is part of the receiver in wireless communication systems and corrects errors which are induced during transmission over the wireless channel. The communications performance of Turbo decoders is determined by Monte Carlo simulation, i.e., the frame error rate (FER) over a large number of frames for different signal-to-noise ratios (SNR) on the wireless channel has to be calculated. Thus, a huge amount

of data has to be processed by the Turbo decoder. Therefore, the implementation of the complete simulation environment directly on the FPGA is an appropriate solution.

Turbo decoding belongs to the class of belief propagation algorithms. It is processed using an iterative algorithm with high computational complexity and high communication bandwidth. This allows exploring and verifying the error protection techniques used in the processors and the interconnect system.

Figure 4 shows the mapping of the communication system (a) on the MPSoC (b). The decoder environment is implemented in the hardware unit called *Turbo slave*. We integrated an additive white Gaussian noise (AWGN) generator from Xilinx [33] for emulating the noisy wireless channel. Turbo decoding runs on the two Leon3 processors. Data is transmitted over the AMBA AHB for reading/writing the input/output data and reading/storing data from/into the SDRAM.

We injected various errors in the different components of the MPSoC. In data and control path, we mimic single event transient and timing errors by generating short pulses in register input signals. This is achieved by adding a multiplexer in front of register input signals. The multiplexer control signal is the injection control generated by a linear phase shift register (LFSR). An LFSR for each bit line is used to generate random bit flips or delays in the data bus signals. An LFSR length of 51 ensures the stochastic independence of the bit flips. The injected error rate (IER) per bit line can be configured at runtime. This allows an efficient design space exploration under varying IER. Three scenarios are emulated to evaluate various error protection techniques:

- In *normal operation*, no errors are injected. This is our reference.
- In *failure operation*, error injection is performed in different components.
- Autonomous error handling in processors and interconnect is activated while *autonomous operation*.

A design space exploration with respect to different error protection scenarios is shown in Figure 5. We used two different codes to protect the transmission of the input data. The first code is a parity bit over all six bits of the input data. In the second code, only the sign bit is doubled. In both cases, the input value is punctured if an error is detected, i.e., it is set to zero which represents a 50:50 probability for a zero and a one. The Turbo decoding algorithm tolerates injected errors up to an IER of almost 10^{-4} without any protection. This demonstrates the algorithmic resilience of Turbo decoding. The decoder refuses completely to work at an IER of 10^{-2} without error protection, whereas the decoding loss with both codes is about 0.6 dB. With an IER of 10^{-3} , the decoding performance with protection by parity bit is nearly equal to the normal operation. Sign duplication performs only about 0.05 dB worse.

Injecting error in data or control path without error protection leads to a complete system failure or the generation of wrong outputs. Whereas, enabling the autonomous error protection results in correct outputs with a very small performance overhead.

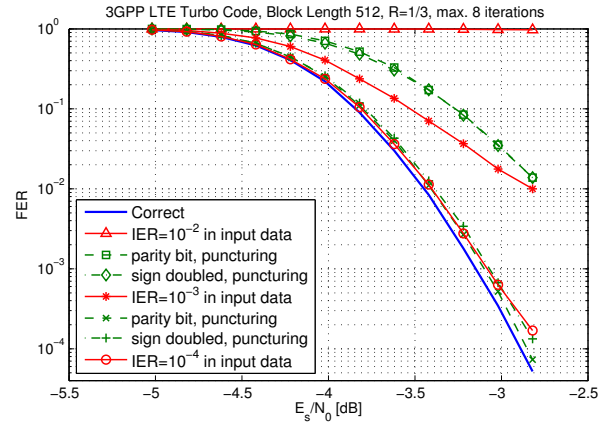


Fig. 5. Example for design space exploration: Error injection in Turbo decoder input data, different IER and protection codes

TABLE III
EXECUTED CLOCK CYCLES AND INSTRUCTIONS OF THE TURBO DECODING SOFTWARE TO SHOW THE ADDITIONAL LATENCY OF THE CPU CORRECTION METHODS. ALSO, THE CLOCK CYCLES PER INSTRUCTIONS (CPI) ARE SHOWN.

operation	clock cycles $\times 10^6$	instruct. $\times 10^6$	inj. errors $\times 10^6$ (IER)	CPI
<i>normal</i>	2575	1386	0 (0)	1.86
<i>auto. CP prot.</i>	3078	1565	37.430 ($\approx 10^{-2}$)	1.97
<i>auto. DP prot.</i>	2918	1421	102.171 ($\approx 10^{-2}$)	2.05

In order to correct errors, some additional clock cycles are necessary as described in the previous sections. The correction in the data path needs two additional clock cycles, whereas the number of needed clock cycles for re-execution of an erroneous control instruction depends on the currently executed instruction. On a simple program counter increment (no control flow instruction) we are able to correct the error in one additional clock cycle, whereas a correction of an erroneous return instruction needs five clock cycles. Furthermore, cache misses due to falsified branch or jump targets have also an impact on the latency.

The clock cycles per instruction (CPI) of the normal operation (no fault injection) and the autonomous operation (with error injection and correction) of the Turbo decoder running on the Leon3 processor is shown in Table III. In the case of the autonomous operation, we see that the mean CPI is higher due to additional clock cycles for correcting errors than in normal operation. Note that all measurements are taken from different runs, and the number of executed instructions vary from run to run.

The investigations show that reliability (in terms of resulting error rates and keeping latency constraints) depends on multiple conditions: application, transient error rate, where the errors occur, and protection technique. An exploration on the microarchitectural level is mandatory for analyzing the impact of transient errors in all parts of the hardware. The presented adaptive prototyping platform allows this exploration by emulating the scenarios under varying constraints and conditions and thus to exploit application level resilience in the design process. A typical emulation of the channel coding

system under injected errors runs *more than 20 hours with two Leon3 cores at a clock frequency of 60 MHz*. In contrast, a software simulation on the microarchitectural level would take years and is thus infeasible. The presented platform is at the best of our knowledge the first rapid prototyping system which provides a holistic combination of techniques for error protection in processors and interconnect that increases the reliability of MPSoCs drastically.

VII. CONCLUSION

In this paper we presented an adaptive system for rapid prototyping and verification of an error-resilient MPSoC architecture. A unique combination of multiple error protection techniques for processor cores and interconnect which have an autonomous behavior with respect to transient errors has been implemented. The demonstration platform allows the fast architectural exploration of various error protection techniques under different failure rates for all signals in the circuit on the microarchitectural level. This is mandatory for exploiting application level resilience in the design process, because simulation including the error injection in hardware is infeasible. With LTE Turbo decoding, a relevant application for state-of-the-art wireless communication was chosen for demonstration.

REFERENCES

- [1] Giovanni De Micheli, "Designing Robust Systems with Uncertain Information," ASP-DAC 2003 Keynote Speech.
- [2] S. V. Adve and P. Sanda, "Reliability-Aware Microarchitecture," *IEEE micro*, vol. 25, no. 6, pp. 8–9, Nov./Dec. 2005.
- [3] *IEEE Design & Test of Computers*, vol. 25, no. 4, July-Aug. 2008.
- [4] M. Nicolaidis, "Time redundancy based soft-error tolerance to rescue nanometer technologies," in *Proc. 17th IEEE VLSI Test Symposium*, 25–29 April 1999, pp. 86–94.
- [5] "Aeroflex Gaisler," <http://www.gaisler.com/>.
- [6] ARM, "AMBA Specification, rev. 2.0, May 1999," <http://www.arm.com>.
- [7] J. George, B. Marr, B. E. S. Akgul, and K. V. Palem, "Probabilistic arithmetic and energy efficient embedded signal processing," in *Proc. of the 2006 international conference on Compilers, architecture and synthesis for embedded systems (CASES '06)*, 2006, pp. 158–168.
- [8] J. Bau, Q. Jacobson, R. Hankins, B. Saha, A. Tabatabai, and S. Mitra, "Error Resilient System Architecture (ERSA) for Probabilistic Applications," in *IEEE Intl. Workshop on Silicon Errors in Logic – System Effects*, apr 2007.
- [9] M. May, M. Alles, and N. Wehn, "A Case Study in Reliability-Aware Design: A Resilient LDPC Code Decoder," in *Proc. IEEE Design, Automation and Test in Europe (DATE '08)*, Munich, Germany, Mar. 2008, pp. 456–461.
- [10] M. Mueller, L. Alves, W. Fischer, M. Fair, and I. Modi, "RAS strategy for IBM S/390 G5 and G6," *IBM J. RES. DEVELOP.*, vol. 43, no. 5/6, 1999.
- [11] H. Inoue, Y. Li, and S. Mitra, "VAST: Virtualization-Assisted Concurrent Autonomous Self-Test," in *Proc. IEEE International Test Conference ITC 2008*, 28–30 Oct. 2008, pp. 1–10.
- [12] B. Hamdi, H. Bederr, and M. Nicolaidis, "A tool for automatic generation of self-checking data paths," in *Proc. 13th IEEE VLSI Test Symposium, 1995*, Apr-3 May 1995, pp. 460–466.
- [13] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proc. 36th International Symposium on Microarchitecture*, December 2003.
- [14] J. Gaisler, "A portable and fault-tolerant microprocessor based on the sparc v8 architecture," in *Proc. International Conference on Dependable Systems and Networks (DSN 2002)*, 2002, pp. 409–415.
- [15] D. J. Lu, "Watchdog processors and structural integrity checking," *IEEE Trans. Computers*, vol. 31, no. 7, pp. 681–685, 1982.
- [16] M. A. Schuette and J. P. Shen, "Processor control flow monitoring using signed instruction streams," *IEEE Trans. Comput.*, vol. 36, no. 3, pp. 264–277, 1987.
- [17] I. Majzik, A. Pataricza, M. Dal Cin, W. Hohl, J. Hönig, and V. Sieh, "Hierarchical checking of multiprocessors using watchdog processors," in *Proc. European Dependable Computing Conference*, 1994, pp. 386–403. [Online]. Available: citeseer.ist.psu.edu/majzik94hierarchical.html
- [18] T. Michel, R. Leveugle, and G. Saucier, "A new approach to control flow checking without program modification," in *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First IEEE International Symposium*, 1991, pp. 334–343.
- [19] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-assisted run-time monitoring for secure program execution on embedded processors," *IEEE Transactions on VLSI Systems*, vol. 14, no. 12, pp. 1295–1308, 2006.
- [20] R. G. Ragel, "Architectural Support for Security and Reliability in Embedded Processors," Ph.D. dissertation, University of New South Wales, Sydney, Australia, 2006.
- [21] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Proc. 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '03)*. Washington, DC, USA: IEEE Computer Society, 2003, p. 581.
- [22] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proc. 12th ACM Conference on Computer and Communications Security (CCS '05)*. New York, NY, USA: ACM Press, 2005, pp. 340–353.
- [23] M. A. Elgamel and M. A. Bayoumi, "Interconnect Noise Analysis and Optimization in Deep Submicron Technology," *IEEE CIRCUITS AND SYSTEMS MAGAZINE*, vol. 3, no. 4, pp. 6–17, 2003.
- [24] S. Krishnamohan and N. Mahapatra, "A highly-efficient technique for reducing soft errors in static CMOS circuits," in *Proc. IEEE International Conference on Computer Design (ICCD 2004)*, 11–13 Oct. 2004, pp. 126–131.
- [25] A. Ejlali, B. M. Al-Hashimi, P. Rosinger, and S. G. Miremadi, "Joint Consideration of Fault-Tolerance, Energy-Efficiency and Performance in On-Chip Networks," in *Proc. 2007 Design, Automation and Test in Europe (DATE '07)*, Apr. 2007.
- [26] S. Murali, T. Theocharides, N. Vijaykrishnan, M. J. Irwin, L. Benini, and G. D. Micheli, "Analysis of Error Recovery Schemes for Networks on Chips," *IEEE Design and Test of Computers*, vol. 22, no. 5, pp. 434–442, 2005.
- [27] F. Worm, P. Jenne, P. Thiran, and G. De Micheli, "On-Chip Self-Calibrating Communication Techniques Robust to Electrical Parameter Variations," *IEEE Design & Test of Computers*, vol. 21, no. 6, pp. 524–535, Nov. 2004.
- [28] D. Bertozzi, L. Benini, and G. De Micheli, "Low power error resilient encoding for on-chip data buses," in *Proc. Design, Automation and Test in Europe Conference and Exhibition*, 4–8 March 2002, pp. 102–109.
- [29] A. Bouajila and et al. , "Organic computing at the system on chip level," in *Proc. International Conference on Very Large Scale Integration of System-on-Chip (VLSI-SoC06)*, Nice, France, Oct. 2006, pp. 338–341.
- [30] Y. Tamir and M. Tremblay, "High-performance fault-tolerant vlsi systems using micro-rollback," in *IEEE Transactions on Computers*, Vol. 39, NO. 4, Oslo, Norway, Apr. 1990.
- [31] D. Ziener and J. Teich, "Concepts for autonomous control flow checking for embedded cpus," in *Proc. 5th International Conference on Autonomous and Trusted Computing (ATC-08)*, Oslo, Norway, Jun. 2008, pp. 234–248.
- [32] D. Ziener and J. Teich, "Concepts for run-time and error-resilient control flow checking of embedded RISC CPUs," *Int. Journal of Autonomous and Adaptive Communications Systems*, vol. 2, no. 3, pp. 256–275, July 2009.
- [33] "Xilinx Inc." <http://www.xilinx.com/>.
- [34] "3GPP LTE Homepage," <http://www.3gpp.org/Highlights/LTE/LTE.htm>.
- [35] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes," in *Proc. 1993 International Conference on Communications (ICC '93)*, Geneva, Switzerland, May 1993, pp. 1064–1070.
- [36] O. Muller, A. Baghdadi, and M. Jezequel, "From Parallelism Levels to a Multi-ASIP Architecture for Turbo Decoding," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 1, pp. 92–102, Jan. 2009.
- [37] F. Naessens, B. Bougard, S. Bressinck, L. Hollevoet, P. Raghavan, L. Van der Perre, and F. Catthoor, "A unified instruction set programmable architecture for multi-standard advanced forward error correction," in *Proc. IEEE Workshop on Signal Processing Systems SiPS 2008*, 8–10 Oct. 2008, pp. 31–36.