Fail-Safety in Core-Based System Design

Baranowski, Rafal; Wunderlich, Hans-Joachim

Proceedings of the 17th IEEE International On-Line Testing Symposium (IOLTS'11)
Athens, Greece, 13-15 July 2011

doi: http://dx.doi.org/10.1109/IOLTS.2011.5994542

Abstract: As scaling of nanoelectronics may deteriorate dependability, fail-safe design techniques gain attention. We apply the concept of fail-safety to IP core-based system design, making the first step towards dependability-aware reuse methodologies. We introduce a methodology for dependability characterization, which uses informal techniques to identify hazards and employs formal methods to check if the hazards occur. The proposed hazard metrics provide qualitative and quantitative insight into possible core misbehavior. Experimental results on two IP cores show that the approach enables early comparative dependability studies.

Preprint

Fail-Safety in Core-Based System Design

Rafal Baranowski, Hans-Joachim Wunderlich
Institute of Computer Architecture and Computer Engineering, University of Stuttgart,
Pfaffenwaldring 47, D-70569 Stuttgart, Germany
{baranowski, wu}@informatik.uni-stuttgart.de

***Abstract*—As scaling of nanoelectronics may deteriorate dependability, fail-safe design techniques gain attention. We apply the concept of fail-safety to IP core-based system design, making the first step towards dependability-aware reuse methodologies. We introduce a methodology for dependability characterization, which uses informal techniques to identify hazards and employs formal methods to check if the hazards occur. The proposed hazard metrics provide qualitative and quantitative insight into possible core misbehavior. Experimental results on two IP cores show that the approach enables early comparative dependability studies.**

***Keywords*—fail-safe design, core-based design, IP reuse methodology**

I. Introduction

The true potential of core-based system design to shorten time-to-market has been enabled by reuse methodologies [1, 2]. Reuse methodologies comprise standard design and verification practices, as well as guidelines for system integration [3]. Core quality is measured with the Quality Intellectual Property (QIP) metric that assesses functionality, configurability, verification completeness, and ease of integration [4].

While aggressive scaling of technology nodes results in increased vulnerability and aging [5, 6], the QIP metric and reuse methodologies in general fail to address dependability. Generic third-party cores cannot directly be used in safety-critical applications and require thorough dependability studies prior to deployment. This incurs additional costs and may involve redesign, compromising the benefits of reuse.

When building a design from reusable cores, the system integrator has no knowledge about possible misbehavior of the third-party cores. This is particularly true for encrypted cores, as the lack of accurate models makes the dependability analysis impossible. At the same time, the actual misbehavior of a failing core and its influence on other system components may be very complex. A single fallible core may considerably compromise the overall system dependability. Considering the increasing vulnerability of nanoelectronics, this clearly calls for dependability-aware reuse methodologies.

To the best of our knowledge, this paper is the first effort towards a methodology for reuse of safety-critical IP cores. We analyze component failures with respect to their impact on the system. A core is considered fail-safe if its failure does not affect correct operation of other system components. We use formal and informal techniques to analyze fail-safety. Results of this analysis are expressed with hazard metrics that enable:

- qualitative statements about fault impact,
- quantitative assessment of fail-safety,
- comparative analysis of alternative designs,
- identification of critical faults and design weaknesses.

The approach presented in this paper serves the purpose of dependability characterization. The analysis is done just once by the core vendor and its results are documented and supplied to the client as part of the deliverables, constituting added value to the product. The system integrator can immediately benefit from the additional information: The proposed metrics enable direct comparison of cores with similar functionality, and facilitate quantitative statements about failure modes. The overall approach meshes well with the widely adopted IP-core reuse practices.

Related Work. While our problem formulation is new in the context of core-based design, similar concerns have been identified in software reuse [7–9]. Our technique builds upon studies that use formal methods to assess design vulnerability to faults [10–13]. These methods verify the full functionality of a core and provide statements about the level of fault tolerance. Rather than proof of fault-tolerance, the aim of our analysis is to verify properties which guarantee safety. We derive hazard metrics using formal property checking on faulty netlists. The faulty netlists are generated exhaustively according to a circuit-level fault model, in contrast to random mutations proposed for evaluation of assertion coverage [14, 15]. Unlike other approaches, our methodology enables early quantitative comparisons of safety for IP cores with similar functionality, without detailed knowledge of the target application.

II. Definitions

The IEC 61058 standard for functional safety of safety-related systems defines *safety* as freedom from unacceptable risk. A failure is said to be *safe* if it does not have the potential to put the safety related system in a hazardous or fail-to-function state. Based on these definitions, fail-safety can be defined as absence of unsafe faults. Although the standard does not formally define fail-safety, it states that such a concept is of value when the failure modes are well defined. In the following, we formally define fail-safety and hazards in the context of core-based design.

A. Fail-safety

We define that a core is fail-safe if, in the event of a failure, all of the following requirements hold:

1) The core does not hinder correct operation of other system components.
2) The core does not impair performance of other system components.
3) If the core is self-testing, its state (healthy/faulty) is reliably communicated to the system components that depend on it.

The definition does not require the entire functionality of a fail-safe core to be maintained in the event of a failure. Instead, it puts limits to the possible misbehavior. While a faulty fail-safe core is not guaranteed to maintain the service it normally provides to the system, it must not disturb other system components: Neither by preventing them from providing their service (first safety requirement), nor by deteriorating their performance (second requirement; relevant for real-time constrained systems). Whenever the service of a fail-safe core is corrupt and the core is self-testing, the failure must be reliably reported to the system (third requirement).

Note that a system composed of fail-safe components is not automaticaly fail-safe or highly reliable. Such a system may still fail if it cannot handle failures of single components. It is the designer's task to assure system operation under single component failures. This task is, however, simplified by the fact that the fault impact is confined to faulty components.

B. Hazard

We define *hazard* as any potential behavior of a core that violates one or more safety requirements. For a given core, the distinction between hazardous and safe behaviors must be made with respect to the role of the device within a system. In the following, we give examples of hazardous and safe behaviors for some typical components of a System on Chip (SoC).

Fig. 1 presents an exemplary SoC. A CPU, together with a JTAG controller is connected to a high-performance bus. A bus bridge connects the high-performance bus to a peripheral bus.

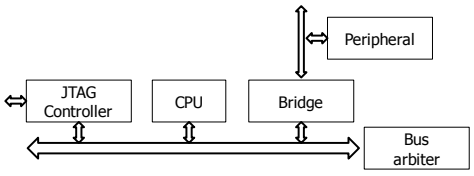

Fig. 1. Example system

Let us assume that the system is spanned by a scan-chain that is controlled by the JTAG controller. While the system operates in the functional mode, the scan chain is deactivated, i.e., the JTAG controller drives the scan enable signal low. In this mode, any erratic behavior of the scan enable signal might corrupt the system state, causing violation of the first safety requirement. Hence, for the JTAG controller we identify that the erroneous assertion of scan enable in the functional mode is a hazard. On the other hand, if a fault impairs debug services of the JTAG controller, it is considered safe as none of the safety requirements is violated.

Let us assume that the only operation performed by the bus bridge is the transport of transactions from the high-performance bus to the peripheral bus. Any data or address corruption caused by a fault in the bridge is considered a hazard, as it might violate the first safety requirement by impairing communication of other system components. Often, for infrastructure components such as the bridge, any deviation from functional specification may be considered a hazard.

The peripheral core in fig. 1 is equipped with just the peripheral bus interface. It cannot hinder other system components, except when it does not comply with the interface specification. The majority of on-chip buses do not have any time-out mechanism, and a deadlock is possible if a slave does not acknowledge a request. In this case, the bus is blocked and other system components cannot communicate. Hence, the lack of acknowledgment from a peripheral core is identified as a hazard.

III. Methodology

In the following, we present a methodology for characterization of core safety. The methodology is composed of four steps: hazard identification (i), formal specification of hazards (ii), core verification under fault condition (iii), calculation of hazard metrics (iv).

A. Hazard identification

In this step, we identify faulty behaviors that have the potential to violate any of the safety requirements. As presently there is no method known that would identify potential hazards automatically, the task is rather challenging. In our approach, we tackle this problem with informal deduction, and – for the sake of clarity and to bring more resolution to the hazard metrics – we use the abstraction of fault-trees.

Fault-tree analysis is a mature analytical technique used to find ways in which an undesired system state can be reached [16]. A fault-tree itself is a graphical model of parallel and sequential combinations of events leading to the *root event*, i.e., an undesired system state. Fault-trees are constructed in a top-down manner, starting from the root event (e.g. unsafe behavior) and listing all contributing events. Dynamic fault-trees [17] are used to model sequential dependency of events. Here, for the sake of brevity, we restrict our discussion to static fault-trees.

For each safety requirement, we construct a fault-tree with violation of the requirement at the root. We then list all events (or logical combinations thereof) that may contribute to the unsafe behavior.

Fig. 2 presents an exemplary fault-tree for a bus bridge with violation of the first safety requirement at the root node. The bridge hinders other system components when it either perturbs communication between other components, or when it does not serve a communication request. Moreover, the

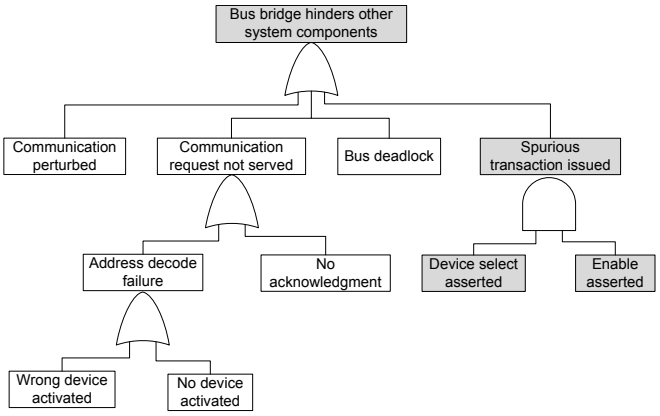

Fig. 2. Fault-tree of a bus bridge

system bus may be blocked by a faulty bridge, and the state of other components may be affected if the bridge issues spurious transactions. The disjunction of the identified events is depicted in the fault-tree with an OR gate. In fig. 2, the tree is developed further for the event of a spurious transaction. Let us assume that the bus interface requires both a device select and an enable signal. In this case, a spurious transaction can only be issued if both the device select and the enable signal are erroneously driven high. This is reflected in the fault tree by an AND gate connected to the two events.

A *cut-set* of a fault-tree is defined as a set of events that are required to occur together to cause the root event (unsafe behavior). A *minimal cut-set* is a cut-set in which all the events are essential for the root event to occur. Each minimal cut-set of the fault-tree is identified as a *hazard*.

In the example, we identify that the simultaneous assertion of the device select and enable signals leads to "spurious transaction", causing the root event to occur. The four events constitute a minimal cut-set (marked gray in fig. 2), which describes a hazard.

B. Formal specification of hazards

In the second step, each identified hazard is expressed formally. To this end, the fault-tree is traversed bottom-up to find logical and temporal properties of hazards. A formal hazard property is expressed with Boolean or temporal logic, or is described with any formal property specification language such as PSL[1].

For instance, let us specify a property for the hazard marked in fig. 2. The hazard occurs only when both the device select (SEL) and enable (EN) are high: $SEL = 1 \wedge EN = 1$. The next level of the fault-tree reveals that the hazard occurs only when no transaction is being requested, as the transaction is spurious ($REQ = 0$). Hence, the hazard is formally captured by the Boolean formula: $SEL = 1 \wedge EN = 1 \wedge REQ = 0$.

[1]http://www.eda.org/ieee-1850

C. Core verification under fault condition

In the third step, we check if the identified hazards occur when the core is subject to faults. A realistic circuit-level fault model is assumed to account for low-level failure mechanisms such as aging and soft-errors. To keep the methodology generic, at this point we do not restrict it to any specific fault model.

For each fault in the circuit-level fault model, a faulty netlist is created. Formal property checking is then employed to check whether the faulty netlist satisfies any hazard property, i.e., if the core is not fail-safe.

While the use of formal techniques guarantees accuracy, their effort is exponential in the worst case. Optimistic estimates of fail-safety can be acquired by fault-injection and simulation of typical workloads, as in [18, 19].

D. Calculation of hazard metrics

The previous step provided a set of circuit-level faults that were proven unsafe. For each hazard, we define its *hazard metric* as the number of unique faults in the circuit-level fault model that were found to potentially trigger the hazard. The *overall hazard metric* is defined as the number of unique faults that caused any hazard in any of the fault-trees. Note that the absolute metric was chosen to enable comparison of hazard metrics for cores of different sizes.

The output of the characterization process consists of:

- The three fault-trees composed for all safety requirements,
- formal specification of hazards,
- hazard metrics, including the overall hazard metric.

E. Discussion

The process of hazard identification and specification requires a considerable effort and knowledge about the core and the interfaces that it should comply with. Moreover, as the proposed approach uses deduction in fault-tree construction, it cannot be guaranteed to identify all possible hazards. A formal, inductive method for specification of hazards from core and interface specifications remains a research challenge. However, as hazard properties depend primarily on specification of interfaces, libraries of hazard properties can be shared for common interfaces and bus architectures. To take full advantage of the methodology, realistic fault models, synthesis libraries, and hazard properties for common buses and core classes should ideally be standardized and shared across different vendors.

IV. Case Study

Within a bus-centric system, all components that constitute the bus infrastructure are very critical, as they may directly prevent other system components from correct operation. For this reason, the case study is based on two AMBA AHB/APB bus bridges:

- `ahb2apb`: A commercial IP core
- `apbctrl`: Part of the Gaisler Research IP (GRIP) library

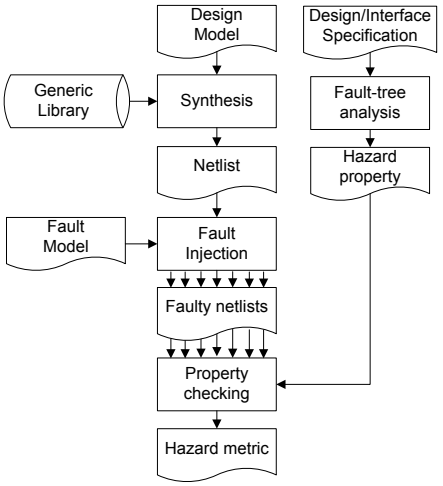

Fig. 3. Characterization flow

The bridges are configured for 16 APB slaves residing at consecutive addresses that are identical in the two cores.

The case study gives an insight into two perspectives: that of the vendor (characterization process), and that of the system integrator (comparative safety study).

A. Circuit-level fault models

Let us define a *design model* D as a set of combinational and sequential components $c \in D$. We assume a generic library consisting of arbitrary single-output components such as logic gates and single-output flip-flops.

Based on the design model D, we define two fault models:

- F_p – permanent fault model – equivalent to the *single stuck-at fault model* with a restriction that a fault $f_c \in F_p$ may only occur on the output of component c (a gate or a flip-flop).
- F_t – transient fault model – assumes that for fault $f_c \in F_t$ the output of component c (or its state in case of a sequential component) is flipped in an arbitrary cycle, and only once in a lifetime.

F_p models defects, whereas F_t addresses soft errors and sporadic timing violations. The hazard metrics derived for F_p provide predictions for yield, whereas the metrics related to F_t give insight into design vulnerability.

B. Experimental setup

The characterization flow is presented in fig. 3. The design models are synthesized for a generic cell library (lsi10k). After synthesis, the examined cores consist of:

- `ahb2apb`: 1467 gates, 107 flip-flops
- `apbctrl`: 1905 gates, 91 flip-flops

The F_t fault model includes one fault for each component, whereas F_p has two faults per component. This gives the following number of faults in the examined cores:

- `ahb2apb`: 1574 in F_t, 3148 in F_p
- `apbctrl`: 1996 in F_t, 3992 in F_p

For each fault in the fault model, a faulty netlist is created according to the rules shown in fig. 4. The inputs of the fault injecting gates (sa0, sa1, spike) constitute additional primary inputs that control fault injection.

We manually construct fault-trees based on the AMBA bus specification and specify hazard properties. Then, formal property checking is performed, employing a commercial property checking tool. The tool is provided with: The faulty netlist, formal hazard properties, as well as fault injection and AMBA-related assumptions expressed formally in PSL language. The total of n runs of the tool is performed, where n is the number of faults in a fault model. For every fault, the tool is used to prove that the fault may trigger a hazard (satisfy the hazard property). For each hazard, we store the list of triggering faults.

C. Hazard properties and assumptions

As the cores have the same functionality and the same interface, they are characterized by the same hazards. Since they constitute bus infrastructure and just mediate in communication between other components, almost any deviation from their functional specification causes violation of the first safety requirement. Through fault-tree analysis, we identified 13 hazards related to the first safety requirement. Due to limited space, hazards that impair performance (second safety requirement) are not considered here. The third safety requirement is skipped as the examined cores are not self-testing. For the sake of brevity, instead of presenting the large fault-tree, in the following we briefly describe the identified hazards:

1) `wrong_data_read`: Upon a valid AHB read transaction, the data forwarded from the APB bus is corrupted.
2) `wrong_data_write`: Upon a valid AHB write transaction, the data forwarded to the APB bus is corrupted.
3) `wrong_decode`: Upon a valid AHB transaction, the corresponding APB slave is not enabled (PSEL signal or PENABLE not asserted).
4) `wrong_address`: Upon a valid AHB transaction, the address forwarded to the APB bus is corrupted.
5) `invalid_enable`: APB enable state is longer than one cycle (PENABLE signal asserted for more than one cycle).
6) `psel_onehot`: More than one APB slave is enabled (the APB PSEL signal vector is not one-hot encoded).
7) `spurious_enable`: APB bus is enabled although no valid transaction was issued on the AHB bus.
8) `read_on_write`: Upon a valid AHB write transaction, an APB read transaction is issued.

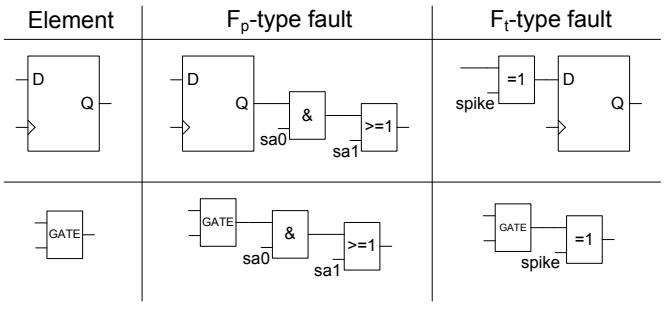

Fig. 4. Fault injection rules

9) `write_on_read`: Upon a valid AHB read transaction, an APB write transaction is issued.
10) `spurious_read`: APB read transaction is issued although no valid AHB read transaction occurred.
11) `spurious_write`: APB write transaction is issued although no valid AHB write transaction occurred.
12) `deadlock`: A valid AHB transaction request is never acknowledged (the HREADY signal is never asserted).
13) `no_service`: A valid AHB transaction is not followed by an APB transaction.

The safety requirements must only hold under the assumption that the environment of a core operates according to specifications. Here, the assumption is that the AHB arbiter, AHB master (initiator) and all APB slaves operate according to the AMBA specification. This assumption is expressed by the following assertions:

1) `C_ahb_decode`: The AHB HSEL signal is asserted only when HADDR signal corresponds to the selected AHB slave.
2) `C_ahb_hsel_onehot`: The AHB HSEL signal vector is either deasserted or one-hot encoded.
3) `C_ahb_stable`: Whenever a new AHB transaction has been issued, the HSEL, HADDR, HTRANS, HWRITE, HSIZE, HBURST and HWDATA signals remain stable until the HREADY signal is asserted.
4) `C_apb_stable`: Whenever an APB slave is enabled, the data on the PRDATA bus is stable.

Additional assumptions are created to control fault injection. In case of an F_t-type fault, the input to the fault-injecting XOR gate (*spike*) receives a positive, one cycle long spike in an arbitrary clock cycle. This is expressed with two formal assumptions: The *spike* signal may be asserted in any cycle, and after this occurs, it is deasserted forever. For an F_p-type fault, the inputs to the fault-injecting OR/AND gate are asserted to "0" or "1", depending on whether a stuck-at-"0" or stuck-at-"1" is to be injected, respectively.

D. Results

The property checking tool was most often able to prove or disprove an assertion in less than 5 seconds on a desktop PC, whereas the per-assertion time limit was set to 10 minutes. The total analysis time of a single core with a single fault model was from 20 to 51 hours. No hazard property was left inconclusive, i.e., all of them were either disproved or proved within the time limit.

Fig. 5 presents characterization results for the two cores based on the transient fault model (F_t). The hazard called `unsafe_behavior` represents the disjunction of all hazards. Each bar indicates a particular hazard metric, whereas the first bar corresponds to the overall hazard metric.

For the `apbctrl` core, out of 1996 F_t-type faults about 71% were proven to trigger at least one hazard (`unsafe_behavior`). Out of these 1417 unsafe faults, more than 86% affect data in read transactions (cause the `wrong_data_read` hazard). This is expected as the bridge contains a large 16-way data multiplexer. Other hazards are much less frequent – by one up to three orders of magnitude. The most serious hazards occur relatively seldom: For 7 faults data is written instead of being read (`write_on_read`), for 3 faults there is a spurious write transaction (`spurious_write`), and just 3 faults cause a deadlock on the AHB bus (`deadlock`). It was proven that no fault in `apbctrl` prevents forwarding of an AHB transaction to the APB bus (`no_service`).

The overall hazard metric (`unsafe_behavior`) for `ahb2apb` is higher by 10%. This means that, with respect to the transient fault model, this core is in general less dependable than `apbctrl`. Moreover, some serious hazards are much more frequent for `ahb2apb`. For instance, address corruption occurs 5 times more often, multiple slaves are enabled 3 times more often, data is written instead of being read 2 times more often, and 2 times more often a spurious write transaction is generated. However, unlike `apbctrl`, the `ahb2apb` core never causes a deadlock on the AHB bus.

Similar comparison is made for the permanent fault model, as shown in fig. 6. The relations between metrics of the two cores are close to those from the previous comparison. A clear exception is the `deadlock` hazard: for F_p it is 3 times more frequent in the `ahb2apb` core.

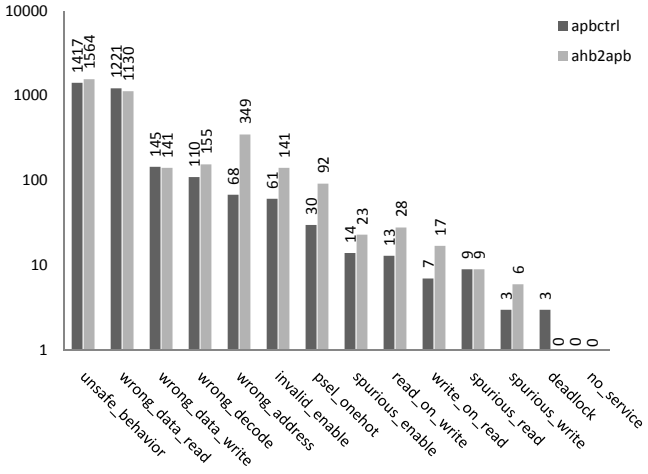

Fig. 5. Number of hazardous F_t faults

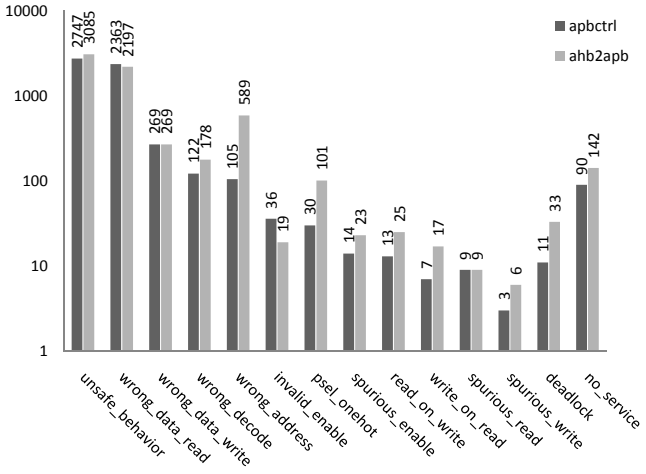

Fig. 6. Number of hazardous F_p faults

Hazard	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)
(1) wrong_data_read	1221	-	-	-	-	-	-	-	-	-	-	-
(2) wrong_data_write	12	145	-	-	-	-	-	-	-	-	-	-
(3) wrong_decode	89	13	110	-	-	-	-	-	-	-	-	-
(4) wrong_address	8	9	25	68	-	-	-	-	-	-	-	-
(5) invalid_enable	43	9	60	25	61	-	-	-	-	-	-	-
(6) psel_onehot	30	0	30	0	30	30	-	-	-	-	-	-
(7) spurious_enable	13	9	14	9	14	0	14	-	-	-	-	-
(8) read_on_write	11	13	13	9	9	0	9	13	-	-	-	-
(9) write_on_read	7	7	6	3	3	0	3	6	7	-	-	-
(10) spurious_read	8	9	9	9	9	0	9	9	3	9	-	-
(11) spurious_write	3	3	3	3	3	0	3	3	3	3	3	-
(12) deadlock	3	0	0	0	0	0	0	0	0	0	0	3

TABLE I
NUMBER OF F_t-TYPE FAULTS TRIGGERING PAIRS OF HAZARDS IN APBCTRL CORE

As ahb2apb contains less logic components than apbctrl and none of the cores contains any explicit fault tolerance mechanisms, it would be reasonable to expect that ahb2apb is safer, i.e., less faults lead to its unsafe behavior. However, our results show the opposite: Even though ahb2apb contains less digital components, its overall hazard metrics are higher. Although more expensive in terms of area, apbctrl is more dependable with respect to both fault models.

Table I shows a matrix representation of the F_t-based characterization results for the apbctrl core. The first column and the first row list the hazards. The rest of the table constitutes a symmetric matrix: Each element in the matrix represents the number of faults that may trigger the two hazards referenced by the row and the column. The elements on the diagonal correspond to the hazard metrics.

The matrix representation allows to study the relationships between hazards. For instance, read_on_write is triggered by 13 faults, which include all faults that trigger both spurious_read and spurious_write, and almost all of the faults triggering write_on_read. If the core is hardened against these 13 faults, four severe hazards can be eliminated.

V. CONCLUSION

We presented a methodology for dependability characterization that augments core-based design with qualification and quantification of safety. We defined fail-safety as the ability of a design to degrade in a way that does not prevent other system components from correct operation. Unsafe behaviors (hazards) are identified using fault-tree analysis. We check if the hazards occur by formal property checking and calculate hazard metrics as the number of faults in a circuit-level fault model that potentially trigger them. Our definition of hazard corresponds to the most critical system failure modes, i.e., problems that may lead to total system failure. The approach is to be followed by core vendors, which are encouraged to derive hazard metrics and supply them to the customer as a standard deliverable.

Experimental results on an AMBA bus bridge showed that, from the perspective of a system integrator, the characterization results provide valuable qualitative and quantitative statements enabling early comparative dependability studies. Hazard metrics explicitly point out the most probable failure modes. They constitute valuable input for system safety studies, and provide justification for mitigation actions.

REFERENCES

[1] R. Gupta and Y. Zorian, "Introducing core-based system design," *IEEE Design Test of Computers*, vol. 14, no. 4, pp. 15–25, 1997.

[2] R. Ranjan, H. Akhiani, Y. Antonioli, C. Deaton, N. Ip, and L. Loh, "Towards harnessing the true potential of ip reuse," in *DesignCon*, 2009.

[3] M. Keating and P. Bricaud, *Reuse methodology manual for system-on-a-chip designs*. Springer, 2002.

[4] K. Werner, "Can IP Quality be Objectively Measured?" in *Design, Automation and Test in Europe*, ser. DATE'04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 330–331.

[5] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, pp. 10–16, November 2005.

[6] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Trans. on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, Sept. 2005.

[7] J. Voas, "Certifying off-the-shelf software components," *Computer*, vol. 31, no. 6, pp. 53–59, Jun. 1998.

[8] D. Hamlet, D. Mason, and D. Woitm, "Theory of software reliability based on components," in *International Conference on Software Engineering*, ser. ICSE'01, May 2001, pp. 361–370.

[9] F. Saglietti, F. Pinte, and S. Sohnlein, "Integration and reliability testing for component-based software systems," in *Software Engineering and Advanced Applications*, ser. SEAA'09, 2009, pp. 368 –374.

[10] R. Leveugle, "A new approach for early dependability evaluation based on formal property checking and controlled mutations," in *IEEE International On-Line Testing Symposium*, ser. IOLTS'05, 2005, pp. 260–265.

[11] S. A. Seshia, W. Li, and S. Mitra, "Verification-guided soft error resilience," in *Design, Automation and Test in Europe*, ser. DATE'07. San Jose, CA, USA: EDA Consortium, 2007, pp. 1442–1447.

[12] G. Fey and R. Drechsler, "A basis for formal robustness checking," in *International Symposium on Quality Electronic Design*, ser. ISQED'08, 2008, pp. 784 –789.

[13] G. Fey, A. Sülflow, and R. Drechsler, "Computing bounds for fault tolerance using formal techniques," in *Design Automation Conference*, ser. DAC'09. New York, NY, USA: ACM, 2009, pp. 190–195.

[14] Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao, "Coverage estimation for symbolic model checking," in *Design Automation Conference*, ser. DAC'99, 1999, pp. 300 –305.

[15] H. Chockler, O. Kupferman, and M. Vardi, "Coverage metrics for formal verification," in *Correct Hardware Design and Verification Methods*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003, vol. 2860, pp. 111–125.

[16] W. Vesely and N. Roberts, *Fault tree handbook*. Nuclear Regulatory Commission, 1987.

[17] J. Dugan, S. Bavuso, and M. Boyd, "Dynamic fault-tree models for fault-tolerant computer systems," *Reliability, IEEE Transactions on*, vol. 41, no. 3, pp. 363 –377, sep 1992.

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

[19] K. Goswami, "Depend: a simulation-based environment for system level dependability analysis," *IEEE Computers*, vol. 46, no. 1, pp. 60 –74, Jan. 1997.