Implications of VLSI Fault Models and Distributed Systems Failure Models — A Hardware Designer's View

Gottfried Fuchs

Vienna University of Technology
Institute of Computer Engineering, Embedded Computing Systems Group
A-1040, Vienna, Treitlstrasse 3/182-2, Austria
fuchs@ecs.tuwien.ac.at

Abstract. The fault and failure models as well as their semantics within the VLSI community and the distributed systems/algorithms community are quite different. Pointing out the mismatch of those fault respectively failure models is the main part of this work. The impact of the failure model in terms of hardware implementation effort and system complexity will be shown for different VLSI implementations of distributed algorithms.
However, still, there are a lot of open questions left, mostly related to the coverage analysis of hardware implemented fault-tolerant algorithms.

Keywords. VLSI, fault models, distributed systems, failure models

1 Introduction

The ever increasing speed, complexity and hence computational power of modern VLSI circuits is mainly enabled due to down-scaling of CMOS technology following *Moore's Law*. Current feature sizes in mass production reached 45nm and it is predicted by the *International Roadmap for Semiconductors 2007 Edition* [1] that CMOS scaling will continue at least down to 22nm.

Taking a closer look at modern and upcoming VLSI technology reveals that decreasing voltage swings and smaller feature sizes — required to achieve high clock frequencies with reasonable power dissipation — renders circuits more and more susceptible to faults. Increasing soft error rates (SER) e.g,. due to particle hits, are becoming an issue not only for circuits in the space and aerospace domain, but also for chips operating at ground level [2].

The above mentioned need for increased attention to faults in VLSI as well as the modular structure of modern Systems-on-Chip (SoCs) — comprising multiple interacting building blocks and several clock domains — interestingly show certain similarities to classical fault-tolerant distributed systems. In the distributed systems community decades of research taking faulty behavior of components into account led to a wealth of concepts, algorithms and fundamental results.

Dagstuhl Seminar Proceedings 08371
Fault-Tolerant Distributed Algorithms on VLSI Chips
http://drops.dagstuhl.de/opus/volltexte/2009/1924

In the DARTS-project (Distributed Algorithms for Robust Tick-Synchronization)[1] our research group investigated how to adopt fault-tolerant algorithms, which have initially been developed and formally analyzed for distributed systems, for VLSI implementations. As a proof of concept a variant of the well-known consistent broadcast primitive by Srikanth and Toueg [3] for generating approximately synchronized clock ticks has been adapted, formally analyzed, implemented and manufactured in $0.18\mu m$ CMOS technology.

1.1 Fault models in VLSI design

Fault models typically applied in VLSI consider faults on the abstraction level of single gates or transistors. A set of interconnected gates forms a circuit. If a gate or interconnect stops to operate properly a fault has occurred (in the context of this work only static fault scenarios are considered). The most commonly treated faults are sketched below.

There are two types of *stuck-at faults*: stuck-at-0/LO and stuck-at-1/HI. A stuck-at-X fault occuring at time t_F at signal S manifests itself in a way that S takes on logic level X at time t_F and is no longer able to change its value after t_F. Note that, if for example signal S is currently LO and a stuck-at-HI occurs, the fault will generate one last erroneous transition to HI and remain there.

A *stuck-open fault* appearing at t_F disconnects the affected signal from its driving buffer which leads to an undefined voltage level, the signal is "floating". This floating state may lead to inconsistent perception of the logic level when read by multiple inputs.

A *delay fault* increases or decreases the time a signal change needs to propagate through the signal/gate. The altered timing behavior of the affected component may lead to the violation of timing constraints in subsequent circuits.

1.2 Failure models in distributed systems

A distributed algorithm typically is composed of one or more tasks operating at each node of the distributed system. Nodes communicate via message passing over links. Single nodes are considered as fault containment regions. Considering the static fault assumption in the VLSI models above, the most appropriate matches in distributed systems failure models seem to be given by component failure models. A subset of these failure models is briefly described in the following paragraphs.

A *crash failure* is present if all tasks at node p properly perform their computing steps and hence, node p correctly emits messages until the crash at time t_F. Starting with t_F, all tasks at the affected node stop to emit any further message — node p has crashed.

An *omission failure* has occurred at t_F if a node omits to send a message m that is scheduled for sending at time t_F.

[1] The project DARTS received funding from the Austrian bm:vit (FIT-IT, contract no. 809456-SCK/SAI).*DARTS project Webpage*: http://ti.tuwien.ac.at/darts/

A *Byzantine failure* is an unrestricted failure type. Therefore, a node affected by a Byzantine failure may show arbitrary malicious behavior. This behavior mainly extends the crash and omission failure models by the possibility to insert additional, inconsistent faulty messages.

1.3 Mapping VLSI faults to failures in distributed systems

The adoption of results from the distributed systems research for problem solving in VLSI design involves **(i)** the mapping of the requirements of a VLSI design (including the fault model to be applied) to a distributed algorithm **(ii)** formal treatment on algorithm level **(iii)** back-transformation of the resulting algorithm to the context of VLSI design.

When intersecting the two very distinct communities some common terms have to be defined. A gate or group of gates in VLSI design will correspond to a computing task in distributed computing. Moreover, the change of the logic level on a signal (e.g. from HI to LO) can be viewed as a message of the distributed algorithm. To be able to choose an appropriate fault-tolerant distributed algorithm for implementation in VLSI the designated fault models have to be defined and mapped to a corresponding distributed systems failure model. Looking at the given definitions of faults in VLSI and failures in distributed systems one can observe a huge gap with respect to the abstraction level. VLSI fault models operate at gate/signal level, while distributed systems failures are considered at the much higher level of nodes.

As stated in its description in Section 1.1 a stuck-at fault may lead to an spurious transition on a signal/rail which is, examined from an algorithms point of view an early timing failure which is NOT covered by crash or omission failures. Hence, the Byzantine failure model represents the only safe model when dealing with stuck-at faults. Similar arguments can be found for stuck-open and delay faults and lead to the same result as for the stuck-at case — only the most unrestricted, Byzantine failure model sufficiently models VLSI faults. Note that, considering additional failure models to the models introduced in Section 1.2 may lead to different results.

2 Fault-tolerant distributed clock generation in VLSI

Algorithm 1 implemented in our DARTS project is, as already noted in the introduction, a variant of the consistent broadcast primitive by Srikanth and Toueg [3]. In a setup of $n \geq 3f+2$ fully connected nodes it generates an approximately synchronized fault-tolerant clock — even in the presence of up to f Byzantine faulty nodes.

Several adaptations had to be made to the original software based algorithm to allow for a VLSI implementation. The most notable change is given by the abstraction of *tick(k)* messages to simple up/down transitions (clock-ticks). This adaptation made it necessary to introduce local buffering of *tick(k)* messages

for every node connected and is represented by the *Remote Pipeline* and *Local Pipeline* in Figure 1 (implemented via elastic pipelines [5]).

The other main building blocks of the implementation are predetermined by the two rules of Algorithm 1, namely the *relay* and the *increment* rules. These two rules have been implemented as m-out-of-$(n-1)$ *Threshold Modules* with $m = f+1$ and $m = 2f+1$ in case of the relay and increment rule respectively. The above mentioned reduction of *tick(k)* messages to simple up/down transitions also made it necessary to separate and therefore duplicate the processing of *up*- and *down*- transitions via the $f+1$ and $2f+1$ threshold modules, leading to a total of 4 threshold modules needed per node.

A more thorough description of the DARTS architecture as well as the formal analysis and proofs can be found in [6] and [7].

In general the implementation requirements of the DARTS Byzantine fault-tolerant clock generation scheme can be summarized as the following: **(i)** A fully connected network of $n \geq 3f+2$ nodes corresponding to $n(n-1)$ links. **(ii)** 4 threshold modules per node $(f+1)$-out-of-$(n-1)$ and $(2f+1)$-out-of-$(n-1)$ (two of each). **(iii)** The threshold modules are the main contributor to chip area even for small numbers of f and n. Moreover, pure digital implementations of m-out-of-$(n-1)$ modules (as it is available with standard CMOS processes) have a very unfavorable scaling of chip area of $\binom{n-1}{f+1}$. For example the increase of tolerable faults from $f = 2$ to $f = 3$ leads to an area increase by a factor of 11.

3 Impact of failure models on VLSI hardware effort

The massive chip area effort for implementation of the Byzantine tolerant algorithm is mostly based on the poor scaling of the threshold modules. Unfortunately, threshold modules prove to be fundamental building blocks when trying to implement fault tolerance and cannot be substituted by other (cheaper) means. Considering the above given facts, the investigation of alternative approaches such as simpler and therefore possibly cheaper algorithms may be promising.

Widder presented in [4] omission- as well as crash-tolerant tick-generation algorithms, Algorithm 2 and Algorithm 3 respectively.

Algorithm 1 Byzantine tolerant tick generation [4]

1: **variables**
2: k : integer := 0
3: **initially** send *tick*(0) to all [once]
// Relay Rule
4: **if** received *tick*(ℓ) from at least $f+1$ remote processes with $\ell \geq k$ **then**
5: send *tick*(k), ..., *tick*(ℓ) to all [once]; $k := \ell$
// Increment Rule
6: **if** received *tick*(k) from at least $2f+1$ remote processes **then**
7: send *tick*($k+1$) to all [once]; $k := k+1$

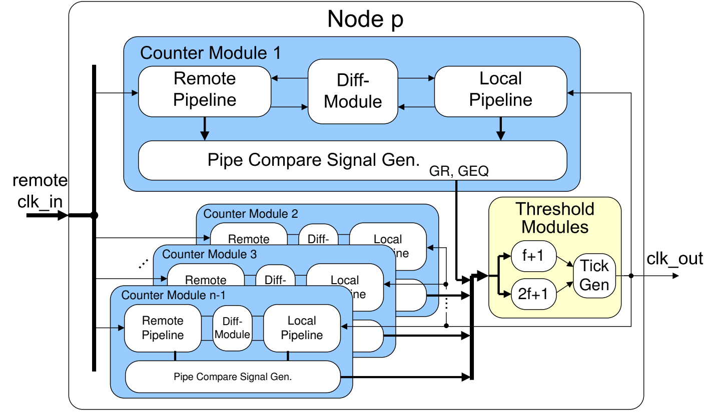

Fig. 1. Hardware implementation of Byzantine tolerant tick-generation.

Algorithm 2 Omission tolerant tick generation

1: **variables**
2: k : integer := 0
3: **initially** send *tick*(0) to all [once]
4: **if** received *tick*(ℓ) from at least 1 remote process with $\ell \geq k$ **then**
5: send *tick*(k), …, *tick*(ℓ) to all [once]; $k := \ell$
6: **if** received *tick*(k) from at least $f + 1$ remote processes **then**
7: send *tick*($k + 1$) to all [once]; $k := k + 1$

Table 1 compares the design properties and hardware effort of Algorithms 1, 2, and 3. It is evident that the simpler algorithms provide substantial savings compared to the Byzantine tolerant implementation. This trend is driven by the simplification and/or reduction of the number of rules and hence smaller threshold modules. Furthermore, theoretical results show that distributed systems built in restricted failure models (like crash or omission) require less nodes for correct operation than systems for the Byzantine case. However, we have to keep the message of Section 1.3 in mind — failure models like crash or omission do **NOT** cover all typical hardware faults included in stuck-at, stuck-open and delay fault models.

Algorithm 3 Crash tolerant tick generation

1: **variables**
2: k : integer := 0
3: **initially** send *tick*(0) to all [once]
4: **if** received *tick*(ℓ) from at least 1 remote process with $\ell \geq k$ **then**
5: send *tick*(k), …, *tick*(ℓ) to all [once]; $k := \ell$

Table 1. Comparison of Algorithms 1, 2 and 3 for $f = 3$

	Byzantine	Omission	Crash
nodes $n \geq$	11	8	5
links	110	53	20
threshold modules per node	4	2+2 (simple)	2 (simple)
chip area per node in μm^2	483859	33188	9810
chip area per system in μm^2	5322449	265504	49050

3.1 Fault coverage discussion

Taking a look at the comparison in Table 1 makes it evident that area effort and system complexity are highly dependent on the chosen failure model underlying the algorithm. Here the question arises, if the price of a factor of ≈ 100 in area effort — when comparing a system composed of crash-tolerant to a system of Byzantine tolerant nodes — is still reasonable, especially when taking into account that more chip area directly translates into more particle hits and hence an increased amount of errors in the system. Admittedly, the Byzantine tolerant algorithm provides a superior coverage on tolerating all types of faults. Investigations on the probabilities and certain properties of faults may provide insights on how to find a good balance between algorithm/implementation complexity and resilience. The most interesting questions in this context are:

"Do those (earlier mentioned) pathological hardware faults which are not covered by crash- or omission-tolerant algorithms occur, and how frequently are they?"

"Is there a fault model less pessimistic than the *Byzantine* one that still covers all hardware related faults?"

4 Conclusions

Implementing fault-tolerant algorithms originally designed for distributed systems in VLSI may be a viable solution to counteract current and upcoming issues concerning SER in VLSI design. Unfortunately, a direct mapping of existing distributed computing results to hardware designs may not be possible for several reasons. The topic discussed here pointed out that a severe mismatch on the abstraction level exists between these two communities when dealing with faulty behaviour. Furthermore, the established fault models in VLSI, as well as the failure models in distributed computing do not translate very well into each other. This leads to the presented result that even a simple stuck-at fault is only covered by the unrestricted Byzantine failure model, which implies very complex and costly algorithms. To which extent the classification of fault types and probabilities of their occurrence can be used to derive a trade-off between hardware effort and fault-coverage still has to be investigated.

Acknowledgements

The presented work evolved from the FIT-IT project DARTS (Distributed Algorithms for Robust Tick-Synchronization), which received funding from the Austrian bm:vit under contract no. 809456-SCK/SAI.
I would like to thank the participants of the Dagstuhl Seminar 08371 on *Fault-Tolerant Distributed Algorithms on VLSI Chips* for the fruitful discussions on the topic. In particular I would like to thank Matthias Függer and Andreas Steininger for their valuable comments on an earlier version of this paper.

References

1. International Technology Roadmap for Semiconductors: (2007)
2. Gadlage, M.J., Eaton, P.H., Benedetto, J.M., Carts, M., Zhu, V., Turflinger, T.L.: Digital device error rate trends in advanced CMOS technologies. Nuclear Science, IEEE Transactions on **53** (2006) 3466–3471
3. Srikanth, T.K., Toueg, S.: Optimal clock synchronization. Journal of the ACM **34** (1987) 626–645
4. Widder, J.: Distributed Computing in the Presence of Bounded Asynchrony. PhD thesis (2004)
5. Sutherland, I.E.: Micropipelines. Communications of the ACM, Turing Award **32** (1989) 720–738 ISSN:0001-0782.
6. Ferringer, M., Fuchs, G., Steininger, A., Kempf, G.: VLSI Implementation of a Fault-Tolerant Distributed Clock Generation. In: Proceedings of the 21st IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT2006), IEEE Computer Society Press (2006) 563–571
7. Fuegger, M., Schmid, U., Fuchs, G., Kempf, G.: Fault-Tolerant Distributed Clock Generation in VLSI Systems-on-Chip. In: Proceedings of the Sixth European Dependable Computing Conference (EDCC-6), IEEE Computer Society Press (2006) 87–96